Tree-walking automata

Mikołaj Bojańczyk ⋆

Warsaw University

Abstract. A survey of tree-walking automata. The main focus is on how the expressive power is changed by adding features such as pebbles or nondeterminism.

1 Introduction

A tree-walking automaton is a sequential device that can recognize properties of trees. The control of the automaton is always located in a single node of the tree; based on local properties of this node, the automaton chooses a new state and moves to a neighboring node. Tree-walking automata have been introduced already in a 1971 paper of Aho and Ullman [1]. The purpose of this paper is to survey the different types of tree-walking automata, with a special focus on expressive power.

A tree-walking automaton can be easily simulated by a branching bottom-up tree automaton, therefore tree-walking automata recognize only regular tree languages. However, the converse inclusion has been a notorious open problem for many years[1]; only recently did [2] establish that tree-walking automata are strictly less expressive than branching automata. Other fundamental properties have also been shown but recently: deterministic tree-walking automata are closed under complement [11], and recognize fewer languages than nondeterministic ones [3]. These results are described in the first part of this survey, although the difficult non-definability proofs are omitted.

A problem with tree-walking automata, and also the reason why they are less expressive than branching automata, is the they easily get lost in a tree. One solution to this problem, due to Engelfriet and Hoogeboom [7], is to allow the automaton to mark tree nodes with pebbles. Although automata with pebbles are still not as strong as branching automata, they form an interesting and robust class of regular tree languages, which is connected to transitive closure first-order logic. A second part of this survey describes the various types of pebble automata, together with recent results on their expressive power.

Most of the proofs here are just informal sketches, intended to give an idea of the type of methods used.

I would like to thank Thomas Colcombet, Anca Muscholl, Damian Niwiński, Luc Segoufin and Balder ten Cate for their helpful comments.

⋆ Supported by Polish government grant no. N206 008 32/0810.

[1] A footnote in the original paper [1] on tree-walking automata states that Michael Rabin has shown that tree-walking automata *do* recognize all regular tree languages.

2 Tree-walking automata

Trees in this paper are binary, labeled, and finite. In other words, each node has either a *left child* and a *right child*, or is a *leaf* with no children at all. Each node also has a *label*, taken from a finite *alphabet* Σ.

A tree-walking automaton is a sequential device that can recognize properties of trees. At any given moment, the automaton is located over a node of the input tree, and assumes one of a finite number of control states. It can do a number of *tests*: "is the current node a leaf?", "is the current node a left (resp. right) child?" and "is the label of the current node a?". Based on the result of these tests, the automaton updates its state, and executes one of the *commands*: "accept the tree", "reject the tree", "go to the parent" and "go the left (resp. right) child". '

A run of the automaton in a given input tree is described as follows. A *configuration* is pair (q, v), where v is a node of the tree, called the *current node*, and q is a *state* taken from a finite state space Q. In such a configuration, the automaton can execute a *transition*, taken from a finite set of allowed transitions, of the form: "if the current state is p and the current node satisfies T (a boolean combination of tests), then execute command C and change the state to q". A *run* is any sequence of configurations that is consistent with the transition table of the automaton. The automaton *accepts* a tree if there is some *accepting run*, i.e. a run that begins in the *initial configuration*—consisting of a designated initial state and the root of the tree—and ends with an "accept" command. Note that a tree may be accepted even if some runs end with a reject command. In particular, the reject command is redundant, but it will be convenient to simplify some of the constructions below. The tree language *recognized* by an automaton is defined to be the set of trees it accepts. A tree-walking automaton is called *deterministic* if in every configuration, there is at most one transition that can applied. Otherwise, the automaton is *nondeterministic.*

Formally, a tree-walking automaton is given as a tuple

$$\mathcal{A} = \langle Q, \Sigma, q_I, \Delta \rangle \ ,$$

where Q is state space, Σ is the input alphabet for labels of tree nodes, $q_I \in Q$ is the designated initial state and Δ is the set of transitions.

We begin with a very simple example automaton, which does a depth-first search through all the tree nodes, and accepts if all nodes have label a:

Example 1. The automaton has three states p, p_{left}, p_{right}. It will visit each non-leaf node three times: for the first time in state p, then in state p_{left} once it has finished inspecting the left subtree, and finally in state p_{right} one it has finished inspecting the right subtree. We only present two sample transitions:

– If the state is p_{right}, the label is a, the node is a left (resp. right) child and not a leaf, then go to the parent and change the state to p_{left} (resp. p_{right}).
– If the state is p_{right}, the label is a and the node is neither a left nor right child (i.e. it must be the root), then accept the tree.

Note how in the above example we test if a node is a left or right child. Wouldn't it be enough to simply have the test "is the current node the root?". As shown by Kamimura and Slutzki in [10], this weaker type of automaton cannot recognize the language "all nodes have label a". Indeed, assume to the contrary, that this language is recognized by an automaton of the weaker type. Consider a balanced binary tree (i.e. all leaves have the same depth) where all nodes have label a. We claim that for every state q and nodes v, w of the same depth, if there is an accepting run of length n that begins in (q, v), then there is also an accepting run of length n that begins in (q, w). This claim is proved by induction on n; the lack of the test is crucial, since otherwise the claim would not work when v would be a right child, and w a left child. If the balanced tree is large enough, the claim implies that any accepting run of this automaton can be modified into one where some leaf is not visited. This unvisited leaf can then be given label b, a contradiction.

Example 2. We will now present a more elaborate example of a tree-walking automaton. The alphabet is $\{\vee, \wedge, 0, 1\}$, and the language consists of the trees that are properly formed logical expressions (i.e. $0, 1$ in the leaves, and $\vee, \wedge$ elsewhere) that evaluate to 1. The idea is to use the following recursive algorithm for evaluating the expression in a subtree, where tail recursion has been removed.

First, evaluate the expression in the left subtree. If the result is 1, and the label in the current node is $\vee$, then the right subtree need not be inspected, and the result 1 can be returned. If the result is 0 and the label in the current node is $\wedge$, then the right subtree need not be inspected, and the result 0 can be returned. Otherwise, the result for the current subtree is the same as the result for the right subtree.

Thanks to the optimization, the above procedure can be realized by a tree-walking automaton. This automaton will have a state p that is used to enter a subtree for the first time, and four other states that will be assumed just after coming back from a subtree. Each of these four states is of the form $p_{j,i}$, where $j \in \{\mathit{left}, \mathit{right}\}$ says which subtree has just been evaluated, and $i \in \{0, 1\}$ is the value of that subtree. We just present two sample transitions of this automaton:

- If the state is $p_{\mathit{left},0}$, the label is $\wedge$, and the current node is a left (resp. right) child, then go to the parent and change the state to $p_{\mathit{left},0}$ (resp. $p_{\mathit{right},0}$).
- If the state is $p_{\mathit{right},i}$, for $i \in \{0, 1\}$, and the current node is a left (resp. right) child, then go to the parent and change the state to $p_{\mathit{left},i}$ (resp. $p_{\mathit{right},i}$).

In the second transition above, the automaton reasons that if it has entered the right subtree, then the value of the left subtree must have been irrelevant.

2.1 Relationship to branching automata

Another automaton model for trees is a *(deterministic bottom-up) branching automaton*. An automaton of this type does a single bottom-up pass through the tree, during which it *evaluates* each tree to a state. The automaton has a finite set of states Q, and a finite set of transitions of two possible forms: "a subtree is evaluated to state q if it has only one node with label a" or "a subtree is evaluated to state q if its root label is a, its left subtree is evaluated to q_0 and its right child is evaluated to q_1". Note how the second transition involves branching, i.e. the subcomputations on the left and right subtrees are done in parallel. The automaton *accepts* a tree if it evaluates it to one of the designated *accepting* states $F \subseteq Q$. We use here the deterministic variant, where the transitions are such that every tree is evaluated to exactly one state. Like for word automata, the subset construction can be used to convert a nondeterministic branching automaton into an equivalent deterministic one.

Branching automata are the standard model for recognizing tree languages, and the name *regular tree language* is applied to tree languages that can be recognized by a branching tree-walking automaton. The class of regular tree languages enjoys all the closure properties of regular word languages, e.g. union, intersection and complementation.

Below we show that tree-walking automata can be compiled into branching automata, and therefore correspond to a subclass of regular languages. As we will see later on, this subclass is proper.

Fact 1 Every tree-walking automaton is equivalent to a branching one.

Proof
The branching automaton will calculate loops. A *loop* is a run that begins and ends in the same node, but not necessarily in the same state. Fix a tree-walking automaton $\mathcal{A}$ with states Q. Given a tree t, let A_t (resp. B_t, C_t) be the set of pairs $(p, q) \in Q^2$ such that $\mathcal{A}$ admits a loop in the root of t with source state p and target state q, assuming that the root of t is treated by the tests in the run as a left child (resp. right child, root). In the above, we only consider loops where the automaton does not try to leave the tree t by doing a "go to the parent" command in the root node. These triples of subsets can be used as states of a bottom-up branching automaton, since the triple assigned to a tree only depends on the root label, and the two triples assigned to its left and right subtrees. ∎

The exponential blowup in the above construction is optimal, for the same reason as for two-way word automata. Consider for instance the set of trees where every label appears either zero or at least two times. This language is recognized by a tree-walking automaton with a state space linear in the size of the alphabet, while any branching bottom-up automaton needs a state space exponential in the size of the alphabet. This example can be modified to work with a one-letter alphabet.

The price for this succinctness is the complexity of emptiness (deciding if the automaton accepts at least one tree). Emptiness for branching automata is a special case of reachability in and-or graphs (the "or" stands for choosing a root label and transition, the "and" stands for having both subtrees accepted), a problem in PTime. This contrasts with tree-walking automata:

Theorem 2
Emptiness for tree-walking automata is ExpTime*-complete.*

Proof
The upper bound follows by translating a tree-walking automaton into a branching automaton of exponential size, and then doing a PTime emptiness test.

The lower bound is similar to the proof that emptiness for two-way word automata is PSpace-hard. For word automata, the key point is that by moving back and forth, a two-way word automaton of $O(n)$ states can detect if two consecutive configurations $c_1 \cdots c_n$ and $d_1 \cdots d_{n\pm1}$ are consistent with the transitions of a Turing machine. For tree-walking automata, we can go from PSpace to ExpTime by encoding the computation tree of an alternating polynomial space Turing machine. Note that the hardness proof works already for deterministic tree-walking automata, or the caterpillar expressions that will be discussed later on. ■

As far as complexity of the emptiness problem is concerned, we lose nothing by adding alternation to tree-walking automata. In an *alternating tree-walking automaton*, sometimes also called an alternating two-way tree automaton, the transitions are the same, only the acceptance mode is changed to a game, played by two players $\forall$ and $\exists$. An alternating tree-walking automaton accepts a tree if player $\exists$ wins the following game. Each game position is a configuration of the automaton in the tree, starting with the initial configuration. To choose who does a move, the state space Q is partitioned into two subsets $Q_\forall$ and $Q_\exists$. When the state belongs to $Q_\forall$, the player $\forall$ chooses a transition and updates the configuration, otherwise this is done by player $\exists$. There are two ways that player $\exists$ can win the game: either when an "accept" command is executed, or when player $Q_\forall$ has no possible transition to apply. In all other cases, which include an infinite play, the player $\forall$ wins.

Alternating tree-walking automata (even on infinite trees, but here the focus is finite trees) can also be compiled into branching automata of exponential size, using a construction similar to the one in Fact 1. Furthermore, alternating tree-walking automata have the same expressive power as branching automata, since alternation can be used to simulate branching.

3 Expressive power

Tree-walking automata are notoriously tricky to analyze. Even what seems like a simple property is non-trivial to prove:

Theorem 3 ([11])
Deterministic tree-walking automata are closed under complementation.

Why is it not enough to swap accept and reject commands? The reason is that the automaton can also reject by entering an infinite loop. Therefore, the above theorem follows from:

Proposition 4 ([11]) Every deterministic tree-walking automaton is equivalent to one where each run ends with either an accept or reject command.

Proof
The idea originates in an observation of Sipser [15]. Let $\mathcal{A}$ be a deterministic tree-walking automaton, with states Q. Without loss of generality, we assume that there is only one transition where the accept command can be used, and this transition requires a state q_F to appear in the root ϵ of the tree. We call (q_F, ϵ) the *accepting configuration*; we can furthermore assume that the accepting configuration admits no other transitions than the accepting one. We will now define an equivalent automaton $\mathcal{B}$, which ends every run by either accepting or rejecting.

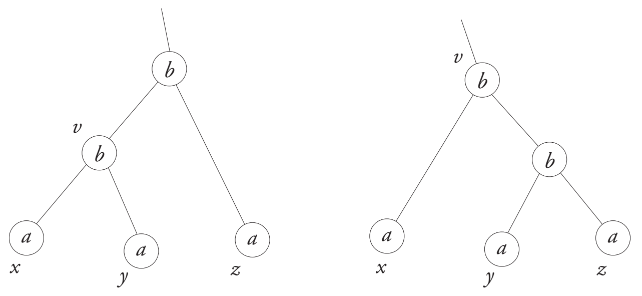
Fix an input tree t. The idea is that the automaton $\mathcal{B}$ does a depth-first search through the *reverse configuration graph* of the automaton $\mathcal{A}$. Vertexes in this graph are configurations of the automaton $\mathcal{A}$ on the input tree t. There is an edge from (p, v) to (q, w) if there is a transition that can take the automaton $\mathcal{A}$ from (q, w) to (p, v). Since the automaton $\mathcal{A}$ is deterministic, each vertex in the configuration graph has indegree at most one. We are only interested in the connected component of the reverse configuration graph that contains the accepting configuration. The key insight of Sipser is that this component is a tree, call it T, and can therefore be efficiently searched in depth-first search manner. The reason is that the accepting configuration has no incoming edges, and hence no loop can appear in T. The simulating automaton $\mathcal{B}$ does a depth-first search on the tree T; if it finds the initial configuration it accepts, otherwise it rejects. To induce a depth-first search, we establish some arbitrary order on the transitions of the automaton $\mathcal{A}$, so that the simulating automaton knows which subtree of T to process first, second, etc. A configuration (p, v) in T is encoded by a configuration in the tree t, with the current node in v. The simulating automaton also needs to remember in its state which subtree of T it has just finished searching. ■

Unfortunately, closure under complementation of deterministic tree-walking automata is a rare instance of good behavior for tree-walking automata. Theorems 5 and 6 below give two instances of bad behavior. A further, only conjectured, instance is:

Conjecture 1. Languages recognized by nondeterministic tree-walking automata are not closed under complementation.

Theorem 5 ([3])
Tree-walking automata cannot be determinized, i.e. there is a tree language L recognized by a nondeterministic tree-walking automaton, but by no deterministic one.

We begin by defining the language L. Consider a tree where all nodes have label b, except for exactly three leaves, which have label a. The *branching structure* is induced by looking at the closest common ancestors of a-labeled leaves, including the a-labeled leaves themselves. (In the above, closest means deepest in the tree.) When there are three a's, there are two possible types of branching structure:



We claim that a nondeterministic tree-walking automaton can tell the difference between the left and right pictures above, while a deterministic one cannot. More formally, we claim that Theorem 5 holds for the following language L:

> All nodes have label b, except for three leaves x, y, z (from left to right), which have label a. The closest common ancestor of x and y is not an ancestor of z.

The negative part of the claim, that any deterministic tree-walking automaton confuses the two types of branching structure, is quite involved and requires developing a pumping argument for tree-walking automata. In this survey, we only present the easier, positive part of the claim. The nondeterministic automaton does five high-level steps (the reader can easily fill in the states and transitions in the actual automaton). Note that only step 3 uses nondeterminism.

1. Doing a depth-first search, check that there are only three nodes labeled a, and these are leaves.
2. Go back to the root. Doing a depth-first search, stop at the first a, i.e. x.
3. Nondeterministically go to some ancestor u of x.
4. Go to the rightmost leaf w below u.
5. Accept if there is exactly one a to the right of w. This can be checked by resuming a depth-first search from the node w.

If the tree belongs to the language, then the automaton will accept, by choosing u in step 3 to be the closest common ancestor v of the leaves x and y. On the other hand, if the tree is outside the language, then either the first step in the automaton will fail, or every possible choice of u will give either zero or two a's to the right of w.

The second result, stated below, shows that even nondeterministic tree-walking automata cannot express some simple regular tree properties.

Theorem 6 ([2])
Tree-walking automata, even nondeterministic ones, do not capture all regular tree languages.

The separating language K is also defined in terms of branching structure. Take a tree over the alphabet $\{a, b\}$, where the label a is only allowed in the leaves, but possibly more than three times. As before, we say a node v is on the *branching* *structure* if it is the closest common ancestor of two leaves with label a. In other words, v is either itself a leaf with label a, or both its left and right subtrees contain each at least one leaf with label a. The language K is defined to contain trees where every leaf with label a has an even number of proper ancestors in the branching structure.

The language K is recognized by a branching automaton with three states. Later on, we will also show that this language can be defined in first-order logic. The proof that K cannot be recognized by any nondeterministic tree-walking automaton is long and complicated, and left out here.

4 Pebble automata

The results above suggest that a tree-walking automaton easily gets lost in a tree; as remarked by Engelfriet and Hoogeboom in [7], "in a binary tree of which all internal nodes have the same label, all nodes look pretty much the same". One way of solving this problem is to add pebbles.

A *pebble automaton*, as defined in [7], is an extended variant of a tree-walking automaton, which can place pebbles on tree nodes during its run. Each pebble automaton has a fixed set of pebbles, which are numbered $1, \ldots, n$. The automaton is defined as a tree-walking automaton, except it has two new types of command: "place pebble i on the current node" and "remove pebble i from the current node"; and a new type of test: "is pebble i on the current node?". There is an important restriction on stack discipline: pebble i can be placed only if pebbles $1, \ldots, i-1$ are on the tree, and pebble i can be removed only if pebbles $i+1, \ldots, n$ are not on the tree. Without the stack discipline, pebble automata would go beyond regular languages, and have undecidable emptiness, even on words and with two pebbles.

A configuration of the pebble automaton is written as $(p, v, v_1, \ldots, v_i)$, where p is the state, v is the current node and $v_1, \ldots, v_i$ are the nodes with pebbles $1, \ldots, i$. Note that the length of the tuple is variable and corresponds to the number of pebbles on the tree. In the initial configuration, no pebbles are placed, the current node is the root, and the state is the initial state.

Example 3. The language K from in Theorem 6 can be defined by a deterministic pebble automaton with one pebble. In particular, pebble automata are more expressive than tree-walking automata. The automaton does a depth-first search traversal of the tree. Whenever it enters a node v (be it for the first, second, or third time), it does the following subcomputation to see if v is on the branching structure: place the pebble on the node v, and then do a depth-first search to test if both the left and right subtrees of v contain leaves with the label a. This way, the automaton can use its finite state memory to know how many (modulo two) ancestors of the currently processed node v are in the branching structure.

It is not immediately obvious that pebble automata recognize regular languages. In a later section on logic, we will show that every pebble automaton can be simulated by a formula of first-order logic with transitive closure. Since the latter logic can only define regular languages, we obtain:

Theorem 7
For every pebble automaton, there is an equivalent branching automaton.

Unfortunately, the coding of pebble automata into branching ones is necessarily non-elementary. This holds already for words. Indeed consider the following sets of words. The set W_1 contains two words a and b. The set W_{n+1} contains all words of the form

$$w_1 a_1 w_2 a_2 \cdots w_k a_k \qquad \text{with } a_1, \ldots, a_k \in \{0, 1\} ,$$

where $w_1 < \cdots < w_k$ are all the words in W_n, ordered lexicographically. The size of the words in these sets grows nonelementarily with n, i.e. with an exponential blowup when passing from n to $n + 1$. Furthermore, one can write a pebble automaton over words with $O(n)$ states and pebbles that accepts the set W_n. As the shortest word recognized by a nonempty pebble-free word automaton is bounded by the number of states, it follows that pebble removal incurs a nonelementary blowup (there is a matching upper bound, i.e. a tower of exponentials of height linear in the number of pebbles). A similar argument can be used to show that the emptiness problem for pebble automata, both for words and trees, is nonelementary. A more precise analysis can be found in [14].

The non-expressivity results for tree-walking automata can be extended to pebble automata:

Theorem 8 ([4])
Pebble automata do not recognize all regular tree languages. Furthermore, for each $n \in \mathbb{N}$, pebble automata with $n + 1$ pebbles recognize strictly more languages than pebble automata with n pebbles. Likewise for deterministic pebble automata.

We should add here that there are two variants of pebble automata in the literature, called *weak* and *strong*. In the weak variant, which is the one defined above, the remove command requires that the removed pebble is on the current node. In the strong variant, there is no such restriction, and a remove pebble command can be executed from any node. Note that even in the strong variant, the stack discipline allows removing only the most recently placed pebble.

Theorem 9 ([4])
Weak pebble automata with n pebbles have the same expressive power as strong pebble automata with n pebbles. Likewise for deterministic pebble automata.

There are two important open problems regarding pebble automata:

1. Is every pebble automaton equivalent to a deterministic one?
2. Are languages recognized by pebble automata closed under complement?

From [4] it follows that there is no constant c such that every k pebble automaton is equivalent to a deterministic one with $c \cdot k$ pebbles. However, determinization may still be possible with a non-linear blowup in the number of pebbles.

An extension of Proposition 4 shows that deterministic pebble automata are closed under complement, see [11]. In particular, a positive answer to the first question would imply a positive answer to the second question. Finally, a positive answer to the second question would imply that pebble automata capture exactly first-order logic with transitive closure, a logic described later on in the paper.

4.1 Invisible pebbles

As Theorem 8 shows, even pebble automata do not have the full expressive power of branching automata. In this section, we present a variant of pebble automata that do.

Fix a branching automaton, with states Q. Consider the following natural algorithm for finding the state to which a tree evaluates. We start at the root. First, we recursively call the algorithm and calculate the state q_0 to which the left subtree evaluates. We place a pebble marked with state q_0 on the left subtree. Then, we recursively call the algorithm and calculate the state q_1 to which the right subtree evaluates. Using the states q_0, q_1 and the label of the root, we can calculate value of the whole tree.

This procedure can be simulated by a pebble automaton, albeit using an unbounded number of pebbles (each with a color from Q). In general, automata with an unbounded number of pebbles, even if these are placed and removed in a stack discipline, have undecidable emptiness. However, the automaton described above has an important property: only the value and position of the most recently placed pebble is ever inspected. This is the motivation for defining automata with invisible pebbles, which were introduced in [9]. The name invisible refers to the fact that all pebbles, except the most recently placed one, are invisible to the automaton.

An *automaton with invisible pebbles* can place an unbounded number of pebbles, furthermore each pebble comes with a color, taken from a finite set C. A configuration of the automaton consists of a head position, a state, as well as a stack $x_1 \cdots x_n$ of pebbles (each pebble is described by its color, and the tree node where it is placed). The stack refers to the times when the pebbles where placed, with x_n being the most recent one. A command "place a new pebble at the current node, with color $c \in C$" extends the stack with a new pebble x_{n+1}, of color c, located in the head position. As with standard (weak) pebble automata, only the pebble on top of the stack can be removed from the tree, and only when the head is over this pebble. The key question is how the pebbles are inspected: the automaton can only test if the current node contains the top pebble on the stack, and if so, what is the color of this pebble. Note that when the automaton removes a pebble, the next newest pebble on the stack becomes visible.

Theorem 10
Automata with invisible pebbles capture exactly the regular tree languages.

Proof
The discussion at the beginning of this section shows how an automaton with invisible pebbles can simulate a branching automaton. For the converse implication, we do only a very rough sketch. We will use alternating tree-walking

automata. (Recall that alternating tree-walking automata recognize exactly the regular tree languages.) An alternating tree-walking automaton can be seen as recognizing a property of nodes, by selecting the nodes from which it admits an accepting run, beginning in the initial state.

Fix an automaton with invisible pebbles. Let p, q be states of the automaton, c a pebble color, and $i \in \mathbb{N}$. In a given tree, we define $\mathit{loop}^i(p, q, c)$ to be the set of nodes v where the automaton can do a loop from state p to state q, assuming that the most recently placed pebble is in the node v and has color c. Furthermore, the loop is not allowed to remove this pebble (and therefore does not depend on the positions of the older pebbles, which remain invisible) and can use at most i new pebbles at any given moment. Note that the same node v describes the position of the top pebble, and the source and target node of the loop.

The key observation is that for any given p, q, c and i, one can write an alternating tree-walking automaton that recognizes the set $\mathit{loop}^i(p, q, c)$. As described above, the alternating automaton is started in a node v, and it accepts whenever v belongs to $\mathit{loop}^i(p, q, c)$. This automaton does not depend on i and is allowed to query membership in sets $\mathit{loop}^{i-1}(p', q', c')$, for various values of p', q', c'. The general idea is that the alternating automaton separately inspects the part of the tree below the node v, and separately inspects the part of the tree above the node v. Therefore, it need not have the position v of the pebble marked on the tree.

When taken over all possible values of p, q, c, these simulating alternating automata can be seen as a transformation on set tuples

$$\{\mathit{loop}^{i-1}(p, q, c)\}_{p,q,c} \quad \mapsto \quad \{\mathit{loop}^{i}(p, q, c)\}_{p,q,c} \ .$$

Since alternating tree-walking automata can simulate fix-points of such transformations, there is an alternating tree-walking automaton for each of the sets

$$\bigcup_{i \geq 0} \mathit{loop}^i(p, q, c) \ .$$

Finally, the sets above contain sufficient information to determine whether a tree gets accepted or not. ■

5 Caterpillar expressions

Caterpillar expressions [5] are to tree-walking automata as regular expressions are to word automata. In other words, caterpillar expressions are an equivalent syntax for tree-walking automata, where the Kleene star is used to replace states.

Fix an alphabet Σ. The *caterpillar alphabet* over Σ consists of two types of letters: commands and tests. The first type of letter is a *command*, used to change the node: `goleft`, `goright` and `goparent`. The second type of letter is called a *test*, these are boolean combinations of the same tests that are allowed in tree-walking automata, which are written `leaf`, `isleft`, `isright` and a, respectively. A word over the caterpillar alphabet is called a *caterpillar word* and describes paths in trees over Σ. For instance, the caterpillar word `isleft` a `goleft` b describes paths that begin in a left-child node with label a, and then go to its left child, which must have label b. If the tree and source node are fixed, a caterpillar word may evaluate to at most one path.

A *caterpillar expression* is a regular set of caterpillar words, given by a regular expression. In a given tree, a caterpillar expression evaluates to a set of paths. A caterpillar expression can be treated as a tree language, by selecting those trees where the expression evaluates to a non-empty set. The above definition could equivalently be restricted to paths that begin and end in the root, since a caterpillar expression if a node is the root.

Proposition 11 Caterpillar expressions define the same tree languages as nondeterministic tree-walking automata.

Proof
As in the Kleene theorem. In the nontrivial part, from tree-walking automata to caterpillar expressions, for every two states p, q of the automaton one defines an expression that describes runs beginning in p and ending in q. ■

What are the caterpillar expressions that correspond to pebble automata? One solution would be to add pebbles to caterpillars. Another, more elegant, solution can be adapted from the work of Segoufin and ten Cate in [16]. The idea is to add a nesting test and a cutting command:

– *Nest.* If c is a caterpillar expression, then $\langle c \rangle$ is a test. This test succeeds in a node v in a tree t if the caterpillar expression c selects at least one path that begins in v.
– *Cut.* There is a new command, called *cut*. This command modifies the whole tree, instead of the current node: it removes all nodes except for the current node and its descendants. In particular, the current node becomes the root, as far as subsequent tests are concerned.

Note that if the caterpillar c does some cutting, the destructive effects are not seen by a caterpillar that uses c in a nested test $\langle c \rangle$. There are two variants of cutting caterpillars. The first is when nesting is only allowed positively, i.e. it cannot be used under the scope of negation. When all nesting is positive, the expression is called a *positive cutting caterpillar*. The unrestricted expressions, where negation of nesting is allowed, are simply called *cutting caterpillars*.

Theorem 12
Positive cutting caterpillar expressions define the same tree languages as pebble automata.

One inclusion is fairly simple: to simulate a positive cutting caterpillar by a pebble automaton. In a preprocessing step, by adding some nesting, the caterpillar is modified so that it issues at most one cut command at each level of nesting. Whenever the cutting command is used, the simulating pebble automaton places a pebble to delimit the cut tree. Whenever a nested test is executed, a pebble is also placed to mark the return point after the test is completed.

The hard part is the converse direction, which can be done using the decomposition lemmas from [4].

What corresponds to cutting caterpillars without the positive restriction? The answer is given in Theorem 15 in the next section, which shows that cutting caterpillars have exactly the same expressive power as first-order logic with transitive closure.

6 Logic

A classical idea in formal language theory, dating at least back to Büchi, is to use logic formulas to define regular languages of words or trees. For trees, a logic formula quantifies over nodes in the tree, and it uses predicates to test labels of these nodes, and structural relationships between these nodes. For instance, the formula

$$\exists x \; \exists y \; \mathit{left}(x,y) \wedge a(x) \wedge b(y)$$

holds in trees where there exist two nodes x, y, such that the node y is a left child of the node x, the node x has label a, and the node y has label y. A formula of logic without free variables, like the one above, defines a tree language: this is the set of trees where the formula is true. In this paper we are interested in formulas that allow binary predicates $x \leq y$, $\mathit{left}(x,y), \mathit{right}(x,y)$ for testing the descendant and left/right child relationship, and for each letter a in the alphabet, a unary predicate $a(x)$ for testing tree labels. From now on, we use the name *first-order logic* for formulas that quantify over nodes and use the above mentioned predicates. An important extension is *monadic second-order logic*, where formulas are additionally allowed to quantify over sets of nodes. A shown in [17], monadic second-order logic has the same expressive power as branching automata, i.e. captures exactly the regular languages. In particular, monadic second-order logic is strictly more expressive than tree-walking automata. What about first-order logic?

Theorem 13 ([2])
The expressive powers of first-order logic and tree-walking automata are incomparable.

Proof
It is not hard to produce a tree-walking automaton recognizing a language that cannot be defined in first-order logic. One example is an automaton that tests if the left-most path has even length, another is an automaton for the language of boolean expressions defined in Example 2. An Ehrenfeucht-Fraïsse argument can be invoked to show that neither of the languages described above can be defined in first-order logic.

The converse result is more surprising, although it had already been conjectured in [6]. It turns out that the language K used in Theorem 6 can be defined in first-order logic. The clever idea, from [13], is that the following language M can be defined in first-order logic: "trees where every leaf has even depth (i.e., an even number of proper ancestors)". We only describe the formula for the language M, from this formula it is not hard to obtain the formula for K. As far as the language M is concerned, there are three types of trees: $M_0 = M$, trees where every leaf has even depth; M_1, trees where every leaf has odd depth; and $M_\perp$, trees where some leaves have even depth, and some leaves have odd depth. There are two observations:

– Take a tree in M_0 (resp. M_1). If we begin in the root and take the path left child, right child, left child, right child, etc., we end up in a leaf that is a right (resp. left) child. Using this observation, one writes a formula φ of first-order logic that is true in all nodes with a subtree in M_0 and false in all nodes with a subtree in M_1.
– If we take a tree in $M_\perp$, and a deepest node x in the tree whose subtree is still in $M_\perp$, then the left subtree of x belongs to M_0 and the right subtree of x belongs to M_1, or vice versa. Therefore, a tree belongs to $M_\perp$ if and only if it has a node with exactly one child that satisfies the formula φ above.

■

If not first-order logic, then what is the appropriate logic for tree-walking automata? We cannot use full monadic second-order logic, since this is too powerful, as Theorems 6 and 8 show. It turns out that the best logic is *transitive closure logic*, which is obtained from first-order logic by adding a transitive closure operator. Let $\varphi(x, y)$ be a formula. By applying the transitive closure operator, we get the formula $(TC_{x,y}\ \varphi)(x, y)$, which is equivalent to the following infinitary disjunction:

$$\bigvee_{i\geq 2} \exists z_1, \ldots, z_i \quad x = z_1 \wedge\ \varphi(z_1, z_2)\ \wedge\ \varphi(z_2, z_3)\ \wedge\ \cdots\ \wedge\ \varphi(z_{i-1}, z_i)\ \wedge\ z_i = y\ .$$

Note that the formula φ may have free variables other than just x, y. These other free variables are also free variables of the transitive closure. Note that here we use only transitive closure of binary relations. For relations of higher arity, transitive closure leads to non-regular languages, a more detailed treatment can be found in [8].

For instance, the following formula defines the ancestor predicate $x \leq y$ in terms of the two children predicates:

$$TC_{x,y}\ \big(x = y \vee \mathit{left}(x, y) \vee \mathit{right}(x, y)\big)\ .$$

Lemma 1. *Every language recognized by a pebble automaton can be defined in transitive closure logic.*

Proof
We first do the proof for tree-walking automata. We claim that for every two

states p, q of the automaton, one can write a formula $\varphi_{p,q}(x, y)$ of transitive closure logic that selects x, y if and only if the automaton admits a run from (p, x) to (q, y). To prove the claim, it is most convenient to compile the automaton into a caterpillar expression, which can easily be translated into transitive closure logic. The translation is by induction on the expression's size, with the transitive closure operation used in place of the Kleene star.

For pebble automata we use the same construction, only we need to extend the formula with variables that mark the locations of the pebbles. Let then $\mathcal{A}$ be a pebble automaton with n pebbles. For $i = 1, \ldots, n$ and states p, q we write

$$(p, x, x_1, \ldots, x_i) \rightarrow (q, y, x_1, \ldots, x_i) \tag{1}$$

if $\mathcal{A}$ admits a run that begins in configuration $(p, x, x_1, \ldots, x_i)$ and ends in configuration $(q, y, x_1, \ldots, y_i)$. Furthermore, this run is not allowed to move any of the pebbles $1, \ldots, i$ along the way, although it may place and remove pebbles $i+1, \ldots, n$ any number of times. We write $\rightarrow^*$ for the transitive closure of $\rightarrow$. We claim that the property in (1) can be expressed by a formula $\varphi_{p,q}(x, y, x_1, \ldots, x_i)$ of transitive closure logic.

The proof of the claim is by induction on $n - i$, with the same construction in both the induction step and base. It is convenient to see (1) as the run of a tree-walking automaton $\mathcal{B}$, which is allowed to do two more kinds of test:

- For $j = 1, \ldots, i$, the automaton $\mathcal{B}$ can ask if pebble j is on the current node.
- If $i < n$, then for any two states r, s of $\mathcal{A}$, the automaton $\mathcal{B}$ can ask if the current node z satisfies

$$(r, z, x_1, \ldots, x_i, z) \rightarrow^* (s, z, x_1, \ldots, x_i, z) \ .$$

Note that the run mentioned in the second test above begins with pebble $i + 1$ placed on the current node, and can be described by a formula of transitive closure thanks to the induction assumption. Since a pebble can be removed only when it is present on the current node, the second type of test can be used to simulate all subruns of $\mathcal{A}$ that involve pebbles $i + 1, \ldots, n$. Whether or not the simulating automaton has a run witnessing (1) can be expressed by a transitive closure formula, using the same ideas as for tree-walking automata. The only difference is that the two new types of tests require using the free variables $x_1, \ldots, x_i$ that describe the positions of pebbles $1, \ldots, i$. This is not a problem, since the formula in the transitive closure operator is allowed to depend on external free variables. ■

A closer look at the above proof reveals that the constructed formula belongs to *positive transitive closure logic*, i.e. the transitive closure operator is not used under the scope of negation. Actually, this is an exact characterization:

Theorem 14 ([7])
Pebble automata have the same expressive power as positive transitive closure logic.

It is worth pointing out that the converse translation, from logic to automata, was done originally for the strong pebble model, where pebbles can be removed even when not placed on the current node. However, thanks to Theorem 9, we know that (weak) pebble automata are the same as strong pebble automata.

By restricting the nesting of the transitive closure operator, one can also obtain a characterization of tree-walking automata without pebbles [12].

What about full transitive closure logic, where there is no restriction on the use of negation? We cite a new, unpublished result:

Theorem 15
Cutting caterpillars have the same expressive power as transitive closure logic. Both are weaker than branching automata.

References

1. A. V. Aho and J. D. Ullman. Translations on a context-free grammar. *Information and Control*, 19:439–475, 1971.
2. M. Bojańczyk and T. Colcombet. Tree-walking automata do not recognize all regular languages. In *ACM Symposium on the Theory of Computing*, pages 234–243, 2005.
3. M. Bojańczyk and T. Colcombet. Tree-walking automata cannot be determinized. *Theoretical Computer Science*, 350(2-3):255–272, 2006.
4. M. Bojańczyk, M. Samuelides, T. Schwentick, and L. Segoufin. Expressive power of pebble automata. In *International Colloquium on Automata, Languages and Programming*, volume 4051 of *Lecture Notes in Computer Science*, pages 157 – 168, 2006.
5. A. Brügemann-Klein and D. Wood. Caterpillars. A context specification technique. *Markup Languages*, 2(1):81–106, 2000.
6. J. Engelfriet, H. Hoogeboom, and J. Van Best. Trips on trees. *Acta Cybernetica*, 14(1):51–64, 1999.
7. J. Engelfriet and H. J. Hoogeboom. Tree-walking pebble automata. In G. Paum J. Karhumaki, H. Maurer and G. Rozenberg, editors, *Jewels Are Forever, Contributions to Theoretical Computer Science in Honor of Arto Salomaa*, pages 72–83. Springer-Verlag, 1999.
8. J. Engelfriet and H. J. Hoogeboom. Automata with nested pebbles capture first-order logic with transitive closure. *Logical Methods in Computer Science*, 3(2:3), 2007.
9. J. Engelfriet, H. J. Hoogeboom, and B. Samwel. XML transformation by tree-walking transducers with invisible pebbles. In *PODS*, pages 63–72, 2007.
10. T. Kamimura and G. Slutzki. Parallel two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51, 1981.
11. A. Muscholl, M. Samuelides, and L. Segoufin. Complementing deterministic tree-walking automata. *Information Processing Letters*, 99(1):33–39, 2006.
12. F. Neven and T. Schwentick. On the power of tree-walking automata. In *International Colloquium on Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, 2000.
13. A. Potthoff. First-order logic on finite trees. In *Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 125–139, 1995.

14. M. Samuelides and L. Segoufin. Complexity of pebble tree-walking automata. In *FCT*, pages 458–469, 2007.
15. M. Sipser. Halting space-bounded computations. In *Foundations of Computer Science*, pages 73–74, 1978.
16. B. ten Cate and L. Segoufin. XPath, transitive closure logic, and nested tree walking automata. Submitted.
17. J. W. Thatcher and J. B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.