Trips on Trees

Joost Engelfriet, Hendrik Jan Hoogeboom, Jan-Pascal Van Best

Department of Computer Science, Leiden University
P.O.Box 9512, 2300 RA Leiden, The Netherlands
e-mail: engelfri@wi.leidenuniv.nl

Abstract. A "trip" is a triple (g, u, v) where g is, in general, a graph and u and v are nodes of that graph. The trip is from u to v on the graph g. For the special case that g is a tree (or even a string) we investigate ways of specifying and implementing sets of trips. The main result is that a regular set of trips, specified as a regular tree language, can be implemented by a tree-walking automaton that uses marbles and one pebble.

1 Introduction

A specification of a function describes the result of the function in terms of its argument. The goal of the programmer is to implement this specification by a program that, for a given argument as input, produces the function result as output. From an elementary point of view, the program can be seen as a device that walks from node u to node v on a graph g. The nodes of g are the possible contents of the program variables, and there is an edge from node m to node n if m can be transformed into n by an elementary programming statement, such as an assignment. The program should find its way through this graph from the initial state u, determined by the input, to the final state v, that determines the output.

In this paper we consider a special case of this general situation, viz. the case that the graph is a finite tree (or even a finite string). In particular, the specification describes a set of triples of the form (t, u, v) where t is a tree (over a ranked alphabet) and u and v are nodes of t. Each such triple can be viewed as a "trip" from u to v on the tree t. Thus, the specification describes a set of trips, i.e., a "trip type". To simplify terminology we will also call this a trip. An example of a trip (type) is: from the left-most leaf to the right-most leaf. This is, of course, the set of all triples (t, u, v) where t is an arbitrary tree, u is its left-most leaf, and v is its right-most leaf.

A set of triples (t, u, v) is said to be *regular* if it forms a regular tree language when, as is quite usual, the nodes u and v are indicated by special marks in t. Thus, a regular trip can be specified by a finite tree automaton (that recognizes the trip) or a regular tree grammar (that generates the trip) or a regular expression (that describes the trip). Moreover, as is well known from [Don,ThaWri] (and from [Büc,Elg] for strings), it can be specified by a formula $\phi(u, v)$ of monadic second-order logic (on trees), with two free variables u and v. Thus, monadic second-order logic is the highest-level specification language for regular trips.

Our main interest is the implementation of regular trip specifications. Given such a specification of trips (t, u, v), we wish to know how we can walk from u to v on t. In other words, we are looking for a general type of automaton that, when started at node u of t can walk to node v along the edges of the tree. Thus, for the trip mentioned above, the automaton should be able to walk to the right-most leaf, whenever it is "dropped" at the left-most leaf (and go into a rejecting state when dropped at any other node).

It is known from [Blo,BloEng2] that, in general, this cannot be done by a finite state tree-walking automaton. The solution in [Blo,BloEng2,BloEng1] is to equip the finite state tree-walking automaton with more powerful tests; in fact, it is allowed to test any property of its current node that can be expressed by a formula (with one free variable) of monadic second-order logic. This is of course not a complete implementation because the tests are still specified in logic. Thus, the question remained how these tests can be implemented. Here we show that regular trips can be implemented by a finite state tree-walking automaton that uses "marbles" and one pebble to find its way through the tree (as Tom Thumb through the forest). The precise way of using marbles and pebble will be explained in Section 4. In Section 3 we start with the easier case of regular trips on strings, and show how to implement them by 2-way finite state automata with one pebble (and no marbles). Section 2 contains the formal definition of a trip.

The results of this paper are part of the Master's Thesis [vBest] of the last author, where more detailed definitions and proofs can be found.

2 Trips and Sites

It should be clear that the reader is assumed to be familiar with formal language theory, and in particular with tree language theory. Thus, the notions of regular tree language and (bottom-up) finite tree automaton are assumed to be good friends of the reader. This can be accomplished by reading [GécSte1] and [GécSte2]. Shame on the reader if he/she did not do so yet!

As explained in the introduction, we are interested in trips on trees. These are now formally defined. Let Σ be a ranked alphabet.

Definition 1. A *trip* is a set of triples (t, u, v) where t is a tree over Σ, and u and v are nodes of t.

Trips go from sites to sites (or from sights to sights?). This is an auxiliary notion that we will need too.

Definition 2. A *site* is a set of pairs (t, u) where t is a tree over Σ and u is a node of t.

To define regular trips (and sites) we first have to show how nodes of trees can be marked. As usual, to code (t, u, v), two booleans are added to the labels of t, one for u and one for v. We define the "marked" ranked alphabet $\mathrm{m}(\Sigma) = \Sigma \cup \{(\sigma, b_1, b_2) \mid \sigma \in \Sigma, b_1, b_2 \in \{0,1\}, (b_1, b_2) \neq (0,0)\}$, where (σ, b_1, b_2) has the same rank as σ. Thus, for each $\sigma \in \Sigma$, $\mathrm{m}(\Sigma)$ contains the symbols σ, $(\sigma, 1, 0)$, $(\sigma, 0, 1)$, and $(\sigma, 1, 1)$. Let t be a tree over Σ and let u and v be nodes of t. We define $\mathrm{mark}(t, u, v)$ to be the tree over $\mathrm{m}(\Sigma)$, with the same nodes and edges as t but with different node labels: Let node x have label σ in t. If x is not u or v, then it also has label σ in $\mathrm{mark}(t, u, v)$; otherwise it has label $(\sigma, x = u, x = v)$. Thus, in $\mathrm{mark}(t, u, v)$, either $u \neq v$ and u is marked by $(1, 0)$ and v by $(0, 1)$, or $u = v$ and it is marked by $(1, 1)$. This defines the coding of trips as tree languages. To code sites as tree languages, we define $\mathrm{mark}(t, u) = \mathrm{mark}(t, u, u)$. Thus, for technical convenience, a site (t, u) gets the same encoding as the "round-trip" (t, u, u). As usual, for a trip T, we define $\mathrm{mark}(T) = \{\mathrm{mark}(t, u, v) \mid (t, u, v) \in T\}$, and for a site S, $\mathrm{mark}(S) = \{\mathrm{mark}(t, u) \mid (t, u) \in S\}$.

Definition 3. A trip T is *regular* if $\mathrm{mark}(T)$ is a regular tree language. A site S is *regular* if $\mathrm{mark}(S)$ is a regular tree language.

As observed in the Introduction, it is well known from the classical results of [Don,ThaWri] that a trip T is regular iff it can be defined by a formula $\phi(x, y)$ of monadic second-order logic, in the sense that T is the set of all (t, u, v) such that $t \models \phi(u, v)$. And similarly for sites and formulas $\phi(x)$ with one free variable.

We will be interested in particular in functional trips, i.e., trips in which the destination is determined by the place of departure. We will show that functional trips can be implemented by deterministic tree-walking automata.

Definition 4. A trip T is *functional* if there are no triples $(t, u, v_1), (t, u, v_2) \in T$ with $v_1 \neq v_2$.

Finally, all the above definitions also apply to the case of strings over an (ordinary) alphabet Σ, with the appropriate changes. Thus, a trip on strings is a set of triples (w, u, v) with $w \in \Sigma^*$ and u, v are positions of w (i.e., $1 \leq u, v \leq |w|$, where $|w|$ is the length of w).

3 Trips on Strings

To find our way on strings we will use 2-way pebble automata. A *2-way pebble automaton* is an ordinary 2-way (nondeterministic) finite state automaton with one pebble. The input string is surrounded by endmarkers on the input tape, and at each moment the automaton is at a certain cell of the tape, in a certain state. It can test the input symbol at the current cell, and move one cell to the left or right, changing state. Additionally, it can drop the pebble on the current cell, it can test whether the pebble is at the current cell, and it can lift the pebble from the current cell (when it lies there, of course). Initially the pebble is not on the input tape, and it is also required that at the end of a computation the pebble is not on the input tape. A 2-way pebble automaton A recognizes a language $L(A)$ in the usual way, but we want to use it to compute a trip, as follows. The *trip* $T(A)$ *computed by* A consists of all triples (w, u, v) such that when A is started at position u of w on the input tape, in its initial state, it can walk to position v, enter a final state, and halt. In other words, if you want to make trip $T(A)$, catch automaton A!

It is well known that 2-way finite automata (without pebble) recognize the regular languages [RabSco,She]. In fact, a 2-way finite automaton A can be simulated by an ordinary (1-way) finite automaton M that, at each cell, computes the *transition table* of A, i.e., the finite set of pairs (q, q') such that when A is started in state q at this cell, it can make an excursion to the left, and return to the cell in state q'. The same technique of transition tables can be used to show that also 2-way pebble automata recognize the regular languages ([BluHew]; cf. [Bir] and Exercise 3.19 of [HopUll]): a 2-way pebble automaton A can be simulated by a 2-way automaton A' (without pebble) that at each cell computes two transition tables of the automaton A (without the instructions that manipulate the pebble), one for excursions to the left and one for excursions to the right. Instead of dropping a pebble on a cell, making excursions to the left and right, and then lifting the pebble again, A' can just stay at the cell and compute A's state change from the two transition tables. From this it is easy to see that every trip computed by a 2-way pebble automaton is regular.

Lemma 5. *For every 2-way pebble automaton A, $T(A)$ is a regular trip.*

Proof. We have to show that $\text{mark}(T(A)) = \{\text{mark}(w, u, v) \mid (w, u, v) \in T(A)\}$ is a regular language. In fact, there is a 2-way pebble automaton A' that recognizes $\text{mark}(T(A))$, i.e., $L(A') = \text{mark}(T(A))$. The automaton A' first walks to u (which is marked by $(1, 0)$ or $(1, 1)$), then simulates a successful walk of A, and finally checks that it is at v (which is marked by $(0, 1)$ or $(1, 1)$). □

Determinism of 2-way pebble automata is defined in the usual way. It should be clear that the trip $T(A)$ computed by a deterministic 2-way pebble automaton A is functional (cf. Definition 4). We now prove that every regular trip can be computed by a 2-way pebble automaton, and in particular by a deterministic automaton if the trip is functional.

Lemma 6. *For every regular trip T on strings there is a 2-way pebble automaton A with $T(A) = T$. Moreover, if T is functional, then A is deterministic.*

Proof. Let M be an ordinary, deterministic finite automaton that recognizes $\text{mark}(T)$. First we describe a nondeterministic automaton A that computes T. The automaton A is started at position u of string w, and it has to walk to position v, with $(w, u, v) \in T$. To do this, A first guesses whether v is to the left or to the right of u, or $v = u$. Suppose that it guesses v to be to the right of u. A drops the pebble at the start position u, walks to the head of the input tape, and then simulates M walking to the right, until it detects the pebble. It picks up the pebble and continues the simulation of M, treating the symbol σ at position

u as $(\sigma, 1, 0)$. Then, nondeterministically, A drops the pebble at some position v, treats the symbol σ at position v as $(\sigma, 0, 1)$, and continues the simulation of M until it reaches the end of the input tape. If M is in a final state, A backs up until it finds the pebble at position v, lifts the pebble, and goes into a final state. In the case that v is to the left of u, A simulates a deterministic finite automaton that recognizes the mirror image of $\mathrm{mark}(T)$, walking from the end to the beginning of the input tape. The case that $v = u$ is obvious.

Let us now assume that T is functional, and describe a deterministic automaton A. It is a variation of the nondeterministic automaton A above. First we argue that A can find out deterministically whether v is to the left or right of u, or at u. Since $\mathrm{mark}(T)$ is a regular language, it should be clear that the language $\{\mathrm{mark}(w, u, v) \mid (w, u, v) \in T \text{ and } v \text{ is to the right of } u\}$ is regular too. Hence, applying the string homomorphism that changes $(\sigma, 0, 1)$ into σ, and $(\sigma, 1, 0)$ into $(\sigma, 1, 1)$, to this language, we obtain that the site $S = \{(w, u) \mid (w, u, v) \in T \text{ for some } v \text{ to the right of } u\}$ is regular. Thus, A can test whether or not v is to the right of u by testing whether or not (w, u) is in site S, and it can do that by dropping its pebble at u and simulating a deterministic finite automaton that recognizes $\mathrm{mark}(S)$, treating the symbol σ at position u as $(\sigma, 1, 1)$. Obviously, A can test in a similar way whether or not v is to the left of u, or at u. Suppose now that v is to the right of u. A then behaves as in the nondeterministic case, simulating M, until it picks up the pebble from u. After that, instead of guessing v nondeterministically, A just tries out all positions v to the right of u, one by one from left to right, moving its pebble from one v to the next. $\square$

Altogether we have proved that the 2-way pebble automaton is the implementation model of regular trips on strings.

Theorem 7. *A trip on strings is regular iff it can be computed by a 2-way pebble automaton. A functional trip on strings is regular iff it can be computed by a deterministic 2-way pebble automaton.*

Since a trip is regular iff it can be expressed in monadic second-order logic, this theorem can be viewed as the generalization from languages to trips of the classical result of Büchi and Elgot [Büc,Elg].

It is shown in [Blo,BloEng2] (for the more general case of trees) that 2-way finite automata cannot compute all regular trips. Thus, the pebble is really needed. We strengthen this result in Theorem 15.

4 Tree-Walking Automata

In the case of strings we have used 2-way automata to walk from one position of a string to another. For trees we need an automaton that walks from one node of a tree to another. Such tree-walking automata were introduced in [AhoUll], and were studied, e.g., in [ERS,KamSlu]. A (nondeterministic) finite state *tree-walking automaton* (or tw automaton, for short) is similar to a 2-way automaton on strings. At each moment the tw automaton is at a certain node of the input tree, in a certain state. It can test the label of the current node, and move to the parent or to one of the children of the node, changing state. A child can be specified by a number between 1 and the rank of the current node label. The automaton can also test whether the current node is the root of the input tree, and if not, what is its "child number", i.e., which child it is of its parent (specified by a number between 1 and the rank of the label of its parent). The language $L(A)$ recognized by a tree-walking automaton A consists of all trees on which A has a computation that starts at the root of the input tree in its initial state, and ends in a final state. As in the case of strings, it can be shown, using the technique of transition tables (for excursions in a subtree), that every tree-walking automaton can be simulated by an ordinary (bottom-up) finite tree automaton. However, as opposed to the case of strings, it is not known whether every finite tree automaton can be simulated by a tree-walking automaton!

Conjecture: The class of tree languages recognized by tree-walking automata is a proper subclass of the regular tree languages.

It should be mentioned here that a statement similar to the one above is proved in [KamSlu]. However, the tree-walking automata of [KamSlu] are weaker than ours: they cannot test the child number of a node; and for this reason, as shown in [KamSlu], they cannot even make a depth-first left-to-right search of the input tree.

Clearly, a type of automaton that can compute all regular trips on trees, should be able to recognize the regular tree languages: for every regular tree language L, $(t, u, u) \mid t \in L, u$ is the root of $t\}$ is a regular trip, and obviously, an automaton that computes this trip also recognizes L. Thus we are led to an automaton that is known to recognize the regular tree languages: the tree-walking marble automaton.

A *tree-walking marble automaton* is a tree-walking automaton that, additionally, can use "marbles" to drop on the nodes of the input tree. The difference between a pebble and a marble is that the automaton has an unlimited supply of marbles (i.e., a marbles bag of infinite size!). Moreover, we want our automaton to have marbles of different colours (which is the reason to call them marbles). Thus, each automaton has a fixed (but arbitrary) number of marble colours, and it has an unlimited supply of marbles of each colour. During its computation, the automaton can drop a marble of a given colour on the current node (provided there is not yet one of that colour), it can test whether a marble of a given colour is at the current node, and it can lift a marble of a given colour from the current node (provided there is one there). Note that there cannot be two marbles of the same colour on a node. There is, however, an important additional restriction on the behaviour of the tw marble automaton: if there are marbles on the current node, then the automaton is not allowed to move up to the parent node. In other words, dropping a marble on a node u closes off the context of u, in the sense that the automaton can only visit u and its descendants, but has to lift all marbles from u to visit the other nodes. Since the automaton starts its computation without marbles on the input tree, this restriction implies that at each moment of time all marbles lie on the path from the current node to the root.

It is shown in [KamSlu] (cf. also [ERS]) that the tw marble automaton recognizes exactly the regular tree languages. However, the model of tw marble automaton is described in a different way in these papers. Instead of marbles, the tree-walking automaton has a pushdown, which has the same length as the path from the current node to the root. The pushdown is synchronized with the movements of the automaton on the tree: a symbol is pushed on the pushdown when the automaton moves to a child, and the top symbol is popped from the pushdown when the automaton moves to the parent. It should be clear that these two types of automata recognize the same tree languages. Each symbol on the pushdown can be simulated by a marble on the corresponding node, taking all pushdown symbols as marble colours. Vice versa, the marbles on the path from the current node to the root can be simulated by a pushdown containing in each cell the colours of the marbles that are on the corresponding node, taking all sets of marble colours as pushdown symbols. The only reason that we have turned the tree-walking pushdown automaton into a tree-walking marble automaton is that the pushdown automaton is not suitable for the computation of trips: when started at a node of the input tree, what would be the content of its pushdown?

The result of [KamSlu] is stated next, together with a sketch of the proof.

Proposition 8. *Both the nondeterministic and the deterministic tw marble automata recognize the regular tree languages.*

Proof. The fact that the language recognized by a nondeterministic tw marble automaton is regular can be proved in the usual way using transition tables (which now also involve marble colours), and we will not go into that.

The other way around we sketch how a deterministic tw marble automaton A can simulate a (deterministic, bottom-up) finite tree automaton M. Let us assume for convenience that the input trees are binary, i.e., that the rank of an input symbol is either 2 or 0. A traverses the input tree t in a depth-first left-to-right fashion, and uses the states of M as marble colours. At each node u of t it determines the state in which M arrives at u (in its own state), as follows. If u is a leaf, it determines M's state from the transition function of M. Otherwise, suppose it has determined the state q_1 at the first child of u. It then drops a marble of colour q_1 on u, walks down to the second child of u, and determines the state q_2 at that child. Moving up to u again, it picks up the marble q_1, and determines the state at u from q_1 and q_2, using M's transition function.

For arbitrary input trees, A uses as marble colours all sequences of states of M of length at most $k-1$ where k is the maximal rank of an input symbol. $\square$

As in the case of strings, to obtain an implementation model for the regular languages an additional pebble is needed. This finally leads us to the main automaton model of this paper: the tree-walking marble/pebble automaton. A *tree-walking marble/pebble automaton* is a tw marble automaton that uses one additional pebble. The pebble can be dropped on a node, detected at a node, and lifted from a node, as usual. Initially and finally, there are no marbles and no pebble on the input tree. However, we need an additional restriction on the behaviour of this automaton, because otherwise non-regular tree languages could be recognized (and hence non-regular trips computed).

Example 9. Consider the non-regular monadic tree language $\{a^n cb^n e \mid n \geq 0\}$, with a, b, c of rank 1 and e of rank 0. This language can be recognized as follows, using the pebble and just one marble: put the marble at the root, and the pebble at the lowest b; then move the marble one node down, and the pebble one node up; repeat this last step, until both the marble and the pebble are at the c-labeled node. □

The additional restriction on the tw marble/pebble automaton is: the pebble can only be dropped or lifted when there are no marbles on the tree. At the end of this section we will discuss a less restrictive definition.

The *trip* $T(A)$ *computed by* a tw marble/pebble automaton A is defined just as in the case of 2-way pebble automata on strings: $T(A)$ consists of all triples (t, u, v) such that when A is started at node u of input tree t, in its initial state, it can walk to node v, enter a final state, and halt. So, for this trip you have to catch marble/pebble automaton A!

We first want to prove that every trip computed by a tw marble/pebble automaton is regular. As in the case of strings (cf. Lemma 5), this easily follows from the fact that the tree languages recognized by tw marble/pebble automata are regular, i.e., that the above restriction has been effective.

We need some terminology on finite tree automata. Let M be a (deterministic, bottom-up) finite tree automaton, and let u be a node of an input tree t. By $\mathrm{state}_{M,t}(u)$ we denote the state in which M arrives at node u. By $\mathrm{succ}_{M,t}(u)$ we denote the set of states q of M such that M arrives in a final state at the root of t when it is started in state q at node u (and such a state is said to be "successful" at u). Thus, for every node u, $t \in L(M)$ iff $\mathrm{state}_{M,t}(u) \in \mathrm{succ}_{M,t}(u)$. We also note that, for a child v of u, $\mathrm{succ}_{M,t}(v)$ can be determined, using M's transition function (for the label of u), from $\mathrm{succ}_{M,t}(u)$ and the states $\mathrm{state}_{M,t}(v')$ for all children $v' \neq v$ of u: to determine whether state p is successful at v, one applies M's transition function to p and all $\mathrm{state}_{M,t}(v')$, and checks whether the resulting state q is successful at u.

Theorem 10. *The tw marble/pebble automata recognize the regular tree languages.*

Proof. By Proposition 8 it suffices to show that every tw marble/pebble automaton A can be simulated by a tw marble automaton A'. Consider a part of a computation of A which starts by dropping the pebble on node u of t, in state q, and ends by lifting it again from u, in state q'. Note that at both moments there are no marbles on t. When simulating A, A' can of course not put a pebble on u (and a marble would not help because it closes off the context of u). Instead it should, somehow, test whether A can make one of the excursions described above, where q is the current state of A and q' is any state of A. In other words, it should be able to test whether (t, u) is in the site $S_{q,q'}$ that consists of all pairs (t, u) such that A can make the excursion described above. We first observe that the site $S_{q,q'}$ is regular. In fact, it is quite clear that $\mathrm{mark}(S_{q,q'})$ can be recognized by a tw marble automaton (and hence is regular by Proposition 8): the automaton first walks to node u which is marked by $(1, 1)$, and then simulates A, starting in state q, treating the mark $(1, 1)$ as the pebble of A, and ending in state q' at u.

Thus, it now suffices to show that a tw marble automaton A' can always be modified in such a way that it can, at each moment, test whether its current node belongs to a given site S. Moreover, the test should be done in a deterministic way because, in the simulation above, several of these sites have to be tested sequentially, viz. $S_{q,q'}$ for all q'. Let M be a bottom-up finite tree automaton that recognizes $\mathrm{mark}(S)$. Note that the input alphabet of M is $\mathrm{m}(\Sigma)$ where Σ is the input alphabet of A'. Clearly, at node u of t, A' can always compute $\mathrm{state}_{M,t}(u)$, using the procedure described in the proof of Proposition 8 (first dropping a marble with a special colour on u, to recognize it after traversing the subtree). Let u have label σ of rank k. To determine whether (t, u) is in S, A' visits the children $u_1, \ldots, u_k$ of u, computes $\mathrm{state}_{M,t}(u_i)$ for every $1 \leq i \leq k$, and returns to u. It then computes $q = \mathrm{state}_{M,\mathrm{mark}(t,u)}(u)$, using M's transition function for the symbol $(\sigma, 1, 1)$. Finally, it checks whether $q \in \mathrm{succ}_{M,t}(u)$. Thus, it remains to explain how the latter test can be implemented. During its computation, A keeps track of $\mathrm{succ}_{M,t}(u)$ by using additional marbles that have the sets of states of M as colours; in particular, there is a marble with colour $\mathrm{succ}_{M,t}(u')$ on every node u' on the path from the current node to the root of t. When A moves from a node u to one of its children v, it can compute $\mathrm{succ}_{M,t}(v)$ (i.e., the colour of the new marble) as described just before this theorem, from $\mathrm{succ}_{M,t}(u)$ (the colour of the marble at u) and the states of M at the other children $v' \neq v$ of u (which it can compute as shown above). When A moves up to the parent of u, it of course first lifts the "succ-marble" from u. Initially A puts a succ-marble with colour F on the root of t, where F is the set of final states of M. □

It is now easy to prove the analogue of Lemma 5 for trees.

Lemma 11. *For every tw marble/pebble automaton A, $T(A)$ is a regular trip.*

Proof. By the previous theorem it suffices to show that there is a tw marble/pebble automaton A' that recognizes $\mathrm{mark}(T(A))$. As in the proof of Lemma 5, A' walks to u, simulates A, and checks that it is at v. □

Next we prove that regular trips can be computed by tw marble/pebble automata.

Lemma 12. *For every regular trip T on trees there is a tw marble/pebble automaton A with $T(A) = T$. Moreover, if T is functional, then A is deterministic.*

Proof. It is shown in [Blo,BloEng1,BloEng2] (using terminology from monadic second-order logic) that regular trips can be computed by tree-walking automata *with regular site tests*, i.e., tw automata that, additionally, have the ability to test whether the current node belongs to a given regular site (for a fixed, but arbitrary number of regular sites). Clearly, a tw marble/pebble automaton can test whether (t, u) is in site S by dropping its pebble on u, checking (according to Proposition 8) whether $\mathrm{mark}(t, u)$ is in the regular tree language $\mathrm{mark}(S)$, with the pebble treated as the mark $(1, 1)$, returning to u, and lifting the pebble. We note that the tw automata with regular site tests are called tw automata *with MSO tests* in [BloEng1,BloEng2].

For the reader who is not familiar with monadic second-order logic, we give a second, direct proof of this lemma (essentially the same as the one in Theorem 8 of [BloEng1] and Theorem 13 of [BloEng2]). Let M be a bottom-up finite tree automaton that recognizes $\mathrm{mark}(T)$. As in the proof of Lemma 6 we first describe a nondeterministic automaton A that computes T. The automaton A is started at node u of tree t, and it has to walk to node v, with $(t, u, v) \in T$. Note that we cannot use the same method as in the proof of Lemma 6; in fact, we cannot pick up the pebble from u during the simulation of M (during a depth-first traversal of the tree), because there will in general be marbles on the tree at that moment. Thus, a more clever simulation is needed. A walks straight from u to v, along the shortest path in t. At each node on that path it uses its pebble and marbles to compute the relevant states of M. First, A guesses whether v is a descendant of u, an ancestor of u, or neither of the two. Suppose that v is neither a descendant nor an ancestor of u. Then A walks up to the least common ancestor z of u and v (which it has to guess) and walks down to v, guessing its way down. On the way up it computes $\mathrm{state}_{M,t'}(x)$ for every node x between u and z, where $t' = \mathrm{mark}(t, u, v)$, and on the way down it computes $\mathrm{succ}_{M,t'}(y)$ for every node y between z and v; finally it computes $\mathrm{state}_{M,t'}(v)$ and checks whether $\mathrm{state}_{M,t'}(v) \in \mathrm{succ}_{M,t'}(v)$. Let us see in more detail how A can do this. It starts by dropping the pebble on u and computing $\mathrm{state}_{M,t'}(u)$, using the procedure in the proof of Proposition 8 and treating the label σ of u as $(\sigma, 1, 0)$. It then lifts the pebble from u, moves one node up, say to x, drops its pebble on x, computes $\mathrm{state}_{M,t'}(u')$ for all children u' of x different from u (and note that this equals $\mathrm{state}_{M,t}(u')$), and applies the state transition function of M to obtain $\mathrm{state}_{M,t'}(x)$. This step is repeated until A arrives at a child, say x_0, of the least common ancestor z. A then computes $\mathrm{succ}_{M,t'}(z)$ (which equals $\mathrm{succ}_{M,t}(z)$) by dropping its pebble on z and, for every state q of M, simulating M on t under the assumption that M has arrived in q at node z (and, of course, lifting the pebble from z after doing this). Let y_0 be the child of z on the path from z to v. Now A computes $\mathrm{state}_{M,t}(w)$ for all children w of z different from x_0 and y_0, and uses these states, together with $\mathrm{state}_{M,t'}(x_0)$ and $\mathrm{succ}_{M,t}(z)$, to compute $\mathrm{succ}_{M,t'}(y_0)$ as indicated just before Theorem 10. Let y be the child of y_0 on the path from y_0 to v. A then computes $\mathrm{state}_{M,t}(y')$ for all children y' of y_0 different from y, and uses them, together with $\mathrm{succ}_{M,t'}(y_0)$, to compute $\mathrm{succ}_{M,t'}(y)$. This step is now repeated until A arrives in v. Finally A computes $\mathrm{state}_{M,t'}(v)$, treating the label σ of v as $(\sigma, 0, 1)$, and checks whether it is in $\mathrm{succ}_{M,t'}(v)$.

The cases that v is a descendant or ancestor of u are similar (A just walks down, or just walks up, respectively). They are therefore left to the reader.

It remains to show that A can be made deterministic if T is a functional trip (cf. Theorem 9 of [BloEng1] and Theorem 14 of [BloEng2]). Note that since the procedure in the proof of Proposition 8 is deterministic, the automaton A only makes nondeterministic moves when there are no marbles or pebble on the tree. Thus, it suffices to show that for such an automaton a deterministic tw marble/pebble automaton A' can be constructed that computes the same trip. Suppose that A is at node w of t in state q, and that A has several possible moves $m_1, \dots, m_k$, of which, of course, at most one is successful, i.e., leads to a final state of A (at the destination v). We claim that A' can find out, for each of these moves m, whether m is successful or not. Consider the site $S_{q,m}$ that consists of all (t, w) such that A has a successful computation on t, starting at node w in state q with move m. Since there is a tw marble/pebble automaton that walks to w and simulates A, starting with move m, it follows from Theorem 10 that $S_{q,m}$ is a regular site. Thus, as explained in the first paragraph of this proof, A' can test whether (t, w) is in $S_{q,m}$, using the (deterministic) procedure in the proof of Proposition 8. ◻

Taking the last two lemmas together we can state the main result of this paper: the tw marble/pebble automaton is the implementation model of regular trips on trees.

Theorem 13. *A trip on trees is regular iff it can be computed by a tw marble/pebble automaton. A functional trip on trees is regular iff it can be computed by a deterministic tw marble/pebble automaton.*

As in the case of strings, since a trip is regular iff it can be expressed in monadic second-order logic, this theorem can be viewed as the generalization of the classical result of Doner and Thatcher/Wright [Don,ThaWri] from tree languages to trips on trees.

As mentioned in the definition of tw marble/pebble automaton, there is a less severe restriction on the behaviour of the automaton that still serves our purposes. To understand this new restriction, we first note that it can always be assumed that there is at most one marble on each node (just take the sets of marble colours as new colours and simulate a set of marbles by one marble). It is easy to see that, under this assumption, the life times of the marbles are *nested*, i.e., included in one another or disjoint from one another; this is due to the fact that a marble closes off the context. Now, in our new definition of tw marble/pebble automaton, rather than requiring that the pebble can only be dropped or lifted when there are no marbles on the tree, we require that the life times of the marbles and the pebble are nested (see [GloHar] for a similar nesting requirement). Intuitively it means that when the pebble is lifted, the "marble configuration" on the tree has to be exactly the same as when it was dropped (and the involved marbles have not been touched in the mean time). It is shown in Theorem 20 of [vBest] that Theorem 10 still holds for these more powerful tw marble/pebble automata, and so does Theorem 13. We note that, under this nesting restriction, the restriction that marbles close off the context cannot be dropped.

Example 14. The non-regular monadic tree language $\{a^n cb^n e \mid n \geq 0\}$ of Example 9 can be recognized as follows, using marbles only, with nested life times. Put a red marble at the root, and a blue marble at the lowest b; then repeat the following step: put a red marble just below the lowest red marble, and put a blue marble just above the highest blue marble. Do this until both the red and blue marble are neigbours of the c-labeled node. Then remove all marbles in the reverse order as they were laid down (i.e., repeatedly the highest blue marble and the lowest red marble). □

We end this paper by showing that Theorem 13 does not hold for tw marble automata, i.e., the pebble is really necessary. Note that it is an open problem whether the marbles are necessary, cf. the Conjecture in the beginning of this section. The proof of the pebble necessity is similar to the one in [Blo,BloEng2] (see Theorem 15 of [BloEng2]).

Theorem 15. *There is a (functional) regular trip on trees that cannot be computed by any (nondeterministic) tw marble automaton.*

Proof. Consider the (monadic) ranked alphabet Σ with symbols b and r of rank 1, standing for "black" and "red", respectively, and one symbol e of rank 0. Let T be the trip consisting of all (t, u, v) such that either t has a red root and v is the root, or t has a black root and v is the child of u (viewing the root as the child of the leaf). Thus, either all trips are to the red root, or everybody visits its child. It should be clear that T is regular. Let us now assume that there is a tw marble automaton A that computes T, and derive a contradiction. The idea is that when A starts at any node u of a tree with a black root, it first has to visit the root to be sure that it is not red. Since there is no way for A to remember its starting point u, A cannot anymore find the child of u. Note that when A is at the root, there are no marbles on the tree, except on the root itself.

Formally, consider the tree $t = b^n e$ with $n > s \cdot 2^c$, where s is the number of states of A and c the number of marble colours. Let $t' = rb^{n-1}e$. Thus, t' is t with its root coloured red. Consider, for every node u of t, the successful walk of A from u to its child. Clearly, during this computation A must visit the root, because otherwise A could make the same computation on t'. As observed above, when A is at the root, all marbles are at the root. Let, at that moment, q_u be the state of A and let M_u be the set of marble(colour)s on the root. Thus, q_u and M_u determine the configuration of A. Hence, by the choice of n, there must be two different nodes u and u' such that, in the corresponding computations, $q_u = q_{u'}$ and $M_u = M_{u'}$ and hence A visits the root in the same configuration in both computations. This implies, however, that A can walk from u to the child of u', a contradiction. □

One may argue that the tw marble/pebble automaton is not a very natural type of automaton, with its rather artificial restrictions on the use of marbles and pebble. The reader is invited to search for a more natural automaton; bread crumbs might be an alternative to marbles and pebbles.

References

[AhoUll] A.V. Aho and J.D. Ullman; Translations on a context free grammar, Inform. and Control 19 (1971), 439–475
cited: 5

[Bir] J.-C. Birget; Two-way automata and length-preserving homomorphisms, Math. Systems Theory 29 (1996), 191–226
cited: 4

[Blo] R. Bloem; *Attribute Grammars and Monadic Second Order Logic*, Master's Thesis, Leiden University, June 1996
`http://www.wi.LeidenUniv.nl/MScThesis/IR96-15.html`
cited: 2, 2, 5, 9, 12

[BloEng1] R. Bloem, J. Engelfriet; Monadic second order logic and node relations on graphs and trees, in Structures in Logic and Computer Science (J.Mycielski, G.Rozenberg, A.Salomaa, eds.), Lecture Notes in Computer Science 1261, Springer-Verlag, 1997, pp.144-161
cited: 2, 9, 10, 10, 11

[BloEng2] R. Bloem, J. Engelfriet; Characterization of properties and relations defined in monadic second order logic on the nodes of trees, Tech. Report 97–03, Leiden University, August 1997
`http://www.wi.LeidenUniv.nl/TechRep/tr97-03.html`
cited: 2, 2, 5, 9, 10, 10, 11, 12, 12

[BluHew] M. Blum and C. Hewitt; Automata on a 2-dimensional tape, in Proc. 8th IEEE Symp. on Switching and Automata Theory, pp.155–160, 1965.
cited: 4

[Büc] J. Büchi; Weak second-order arithmetic and finite automata, Z. Math. Logik Grundlag. Math. 6 (1960), 66–92
cited: 1, 5

[Don] J. Doner; Tree acceptors and some of their applications, J. of Comp. Syst. Sci. 4 (1970), 406–451
cited: 1, 3, 11

[Elg] C. C. Elgot; Decision problems of finite automata and related arithmetics, Trans. Amer. Math. Soc. 98 (1961), 21–51
cited: 1, 5

[ERS] J. Engelfriet, G. Rozenberg, G. Slutzki; Tree transducers, L systems, and two-way machines, J. of Comp. Syst. Sci. 20 (1980), 150-202
cited: 5, 7

[GécSte1] F. Gécseg, M. Steinby; *Tree automata*, Akadémiai Kiadó, Budapest, 1984
cited: 2

[GécSte2] F. Gécseg, M. Steinby; Tree Languages, in G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, *Volume 3: Beyond Words*, Chapter 1, Springer-Verlag, 1997
cited: 2

[GloHar] N. Globerman, D. Harel; Complexity results for two-way and multi-pebble automata and their logics, Theor. Comput. Sci. 169 (1996), 161–184
cited: 11

[HopUll] J.E. Hopcroft, J.D.Ullman; *Introduction to Automata Theory, Languages and Computation*, Addison–Wesley, Reading, Massachusetts, 1979.
cited: 4

[KamSlu] T. Kamimura, G. Slutzki; Parallel and two-way automata on directed ordered acyclic graphs, Inf. and Control 49 (1981), 10–51
cited: 5, 6, 6, 6, 7, 7

[RabSco] M.O. Rabin, D. Scott; Finite automata and their decision problems, IBM J. Res. Devel. 3 (1959), 115–125
cited: 4

[She] J.C. Shepherdson; The reduction of two-way automata to one-way automata, IBM J. Res. Devel. 3 (1959), 198–200
cited: 4

[ThaWri] J. W. Thatcher, J. B. Wright; Generalized finite automata theory with an application to a decision problem of second-order logic, Math. Systems Theory 2 (1968), 57–81
cited: 1, 3, 11

[vBest] J.P. van Best; *Tree-Walking Automata and Monadic Second Order Logic*, Master's Thesis, Leiden University, July 1998
`http://www.wi.LeidenUniv.nl/MScThesis/IR98-06.html`
cited: 2, 11