Draft of a paper to appear in IPSJ-SIGPRO.

Checking Time Linearity of Regular Expression Matching Based on Backtracking

Satoshi Sugiyama[1,a)] Yasuhiko Minamide[1,b)]

Abstract: Most implementations of regular expression matching in programming languages are based on backtracking. With this implementation strategy, matching may not be achieved in linear time with respect to the length of the input. In the worst case, it may take exponential time. In this paper, we propose a method of checking whether or not regular expression matching runs in linear time. We construct a top-down tree transducer with regular lookahead that translates the input string into a tree corresponding to the execution steps of matching based on backtracking. The regular expression matching then runs in linear time if the tree transducer is of linear size increase. To check this property of the tree transducer, we apply a result of Engelfriet and Maneth. We implemented the method in OCaml and conducted experiments that checked the time linearity of regular expressions appearing in several popular PHP programs. Our implementation showed that 47 of 393 regular expressions were not linear.

Keywords: regular expression, tree transducer, linear size increase

1. Introduction

Regular expressions are extensively used in programs that manipulate strings such as web programs. However, there is an issue that regular expression matching may take an unexpectedly long time. In the worst case, it may take exponential time with respect to the length of the input. This is because most implementations of regular expression matching in programming languages are based on backtracking. Recently, it has been pointed out that this issue can be exploited for DoS attack [14]. Furthermore, this may not only be just a matter of time, but may also change the behavior of a program unexpectedly in implementations such as PCRE (Perl-compatible regular expression) [10]. PCRE involves a limit on the number of steps executed for regular expression matching. If the number of steps exceeds this limit, the matching is considered to fail even if it would succeed eventually without the limit [10] [*1].

In this paper, we develop a method to check whether regular expression matching runs in linear time with respect to the length of the input. This method is significant for the following situations.

- The method can be used as a criterion for selecting a particular implementation of regular expression matching. Although regular expression matching based on backtracking takes exponential time in the worst case, its execution is fast for expressions with linear execution time. It would then be possible to choose an implementation based on backtracking for those expressions with linear execution time.
- The method can be used to check existing programs involving regular expression matching. We could then ascertain that regular expression matching runs in a reasonable amount of time and does not exceed the limit. If we were to find regular expressions whose matching may not run in linear time, we could then devise a more efficient equivalent regular expression, or use another implementation not based on backtracking.

Our method constructs a tree transducer from a regular expression and then applies the theory of tree transducers to check its properties. We construct a top-down tree transducer with regular lookahead that translates an input string into a tree corresponding to the execution steps of matching based on backtracking. The size of the output tree is proportional to the running time for regular expression matching. Therefore, the regular expression matching runs in linear time if the size of every output tree of the tree transducer is linearly bounded by the size of the input tree. We say that a transducer is of *linear size increase* if it has this property. It is decidable whether a tree transducer is of linear size increase by checking whether an equivalent input proper tree transducer is *finite copying*. The notion of finite copying was introduced by Aho and Ullman [1] and it says that any position of an input is translated by a transducer boundedly many times. To check this property, we apply a result of Engelfriet and Maneth [7].

We have implemented the method in OCaml and conducted experiments that checked the time linearity of regular expressions appearing in several popular PHP programs. The experiments showed that 47 of 393 regular expressions were not linear. It should be noted that our implementation only checks the property of finite copying. Although linear size increase does not imply finite copying in general, we believe that finite copying is a necessary and sufficient condition for linear size increase for transducers constructed from regular expressions by our method.

[1] Department of Computer Science, University of Tsukuba
[a)] sugiyama@score.cs.tsukuba.ac.jp
[b)] minamide@cs.tsukuba.ac.jp

[*1] Whether the limit has been exceeded can be checked by calling the function 'preg_last_error()', which returns the latest error code in the PCRE. However, many programs do not check the error code.

2. Preliminaries

The set of natural numbers is denoted by $\mathbb{N}$. For $k \in \mathbb{N}$, $[k]$ denotes the set $\{1, \ldots, k\}$. For a set S, $|S|$ is the cardinality of S.

For a relation $r \subseteq S \times T$ and an element $s \in S$, $r(s) = \{t \in T \mid (s,t) \in r\}$. For $S' \subseteq S$, $r(S') = \bigcup_{s \in S'} r(s)$. The domain and range of r are defined by $\mathrm{dom}(r) = \{s \in S \mid r(s) \neq \emptyset\}$ and $\mathrm{range}(r) = r(S)$. For $r_1 \subseteq S \times T$ and $r_2 \subseteq T \times U$, their composition is $r_1 \circ r_2 = \{(s,u) \in S \times U \mid (s,t) \in r_1, (t,u) \in r_2\}$.

A ranked alphabet is a tuple (Σ, Arity) where Σ is a finite set and Arity is a function of type $\Sigma \to \mathbb{N}$. For $\sigma \in \Sigma$, $\mathit{Arity}(\sigma)$ is the arity of σ. If we write $\sigma^{(n)}$, it implicitly requires $\mathit{Arity}(\sigma) = n$. $\Sigma^{(n)}$ denotes $\{\sigma \mid \sigma^{(n)} \in \Sigma\}$.

A structure constructed from a ranked alphabet is called a ranked tree and we usually call it just a tree. The set of trees constructed by a ranked alphabet Σ is denoted by T_Σ. A subset of T_Σ is called a tree language. Let X be a set of variables. Then, $T_\Sigma(X)$ is the set of trees constructed from Σ and X. $V(t)$ denotes the set of nodes of a tree t and is inductively defined by $V(\sigma(t_1, \ldots, t_k)) = \{\epsilon\} \cup \{iu \mid u \in V(t_i), i \in [k]\}$ where ϵ represents the root of the tree. For a tree t, the size of t is defined by $\mathrm{size}(t) = |V(t)|$. For a node u, t/u denotes the subtree of t at u. It is defined by $t/\epsilon = t$ and $\sigma(t_1, t_2, \ldots, t_k)/iu = t_i/u$. The lexicographic order over the nodes of t is denoted by $\leq$. For $t \in T_\Sigma(X)$, $t_1, \ldots, t_k \in T_\Sigma$, $x_1, \ldots, x_k \in X$, we denote by $t[x_i \leftarrow t_i \mid i \in [k]]$ the tree obtained by substituting t_i for x_i in t.

We define two representations of strings over Σ as trees. In $\mathit{MON}(\Sigma)$, every $\sigma \in \Sigma$ has rank 1 and an additional symbol with rank 0 is used to denote the end of a string. In $\mathit{STR}(\Sigma)$, every $\sigma \in \Sigma$ has rank 0 and two additional symbols are used: a symbol denoting the empty string and a symbol with rank 2 denoting the concatenation of two strings. That is,

$$\begin{aligned} \mathit{MON}(\Sigma) &= \{\sigma^{(1)} | \sigma \in \Sigma\} \cup \{\epsilon_M^{(0)}\}, \\ \mathit{STR}(\Sigma) &= \{\sigma^{(0)} | \sigma \in \Sigma\} \cup \{\bullet^{(2)}, \epsilon_S^{(0)}\}. \end{aligned}$$

For example, the string abc is represented as $a(b(c(\epsilon_M)))$ over $\mathit{MON}(\Sigma)$ and $\bullet(a, \bullet(b,c))$ over $\mathit{STR}(\Sigma)$. In the case of $\mathit{MON}(\Sigma)$, strings and trees are in one-to-one correspondence. In the case of $\mathit{STR}(\Sigma)$, their correspondence is one-to-many. For example, the string abc can be represented as $\bullet(\bullet(a,b),c)$ or $\bullet(\bullet(a,\epsilon_S),\bullet(b,c))$ in addition to the representation above. For $\mathit{STR}(\Sigma)$, we say t is *compact* if it is either ϵ_S or a tree that does not contain ϵ_S.

3. Tree-based Semantics of Regular Expression Matching

We focus on regular expression matching with priorities compatible with Perl and represent the execution steps of regular expression matching by a tree. The size of the tree is proportional to the running time of the matching. Therefore, we can decide its running time from the size of the tree. The semantics of regular expression matching is given by the nondeterministic parser of Sakuma et al. [15] defined by using the list monad.

The syntax of regular expressions is standard and defined as follows.

$$\begin{array}{rcll} r & ::= & \epsilon & \text{(empty string)} \\ & | & c & \text{(symbol)} \\ & | & r_1 r_2 & \text{(concatenation)} \\ & | & r_1 | r_2 & \text{(alternation)} \\ & | & r_1^* & \text{(repetition)} \end{array}$$

The language of a regular expression r, denoted by $L(r)$, is defined as follows.

$$\begin{aligned} L(\epsilon) &= \{\epsilon\} \\ L(c) &= \{c\} \\ L(r_1 r_2) &= L(r_1) L(r_2) \\ L(r_1 | r_2) &= L(r_1) \cup L(r_2) \\ L(r_1^*) &= L(r_1)^* \end{aligned}$$

For the purposes of this discussion, we exclude regular expressions that contain a subexpression r_1^* such that $\epsilon \in L(r_1)$. This is because such expressions prevent a simple and intuitive definition of the semantics of Sakuma et al. [15] below, making them nonterminating and ill defined. Furthermore, such regular expressions are rarely used in practice.

Regular expressions in programming languages have priorities in matching. In the alternation $r_1|r_2$, r_1 has a higher priority than r_2. If both of them match, the string is considered as matched with r_1. Repetition r_1^* is expanded as $r_1^* = r_1 r_1^* | \epsilon$ and it therefore matches as many repetitions of substrings matching with r_1 as possible.

The semantics is defined so that a result of matching with a higher priority comes earlier in the list. The concatenation of two lists is denoted by $+\!\!+$. The semantics of regular expression matching is defined as follows.

Definition 3.1 (Semantics of regular expression matching)**.**

$$\begin{aligned} N[\![r]\!] &:: \Sigma^* \to \Sigma^* \; \mathit{list} \\ N[\![\epsilon]\!]w &= [w] \\ N[\![c]\!]w &= \begin{cases} [w'] & \text{if } w = cw' \\ [] & \text{otherwise} \end{cases} \\ N[\![r_1 r_2]\!]w &= N[\![r_1]\!]w \gg\!= \lambda w'.N[\![r_2]\!]w' \\ N[\![r_1|r_2]\!]w &= N[\![r_1]\!]w +\!\!+ N[\![r_2]\!]w \\ N[\![r_1^*]\!]w &= (N[\![r_1]\!]w \gg\!= \lambda w'.N[\![r_1^*]\!]w') +\!\!+ [w] \end{aligned}$$

$N[\![r]\!]w$ returns a list of all possible strings that are obtained by consuming a prefix of w matching with r.

Although the above semantics shows how to run regular expression matching, it explicitly says nothing about the execution steps of regular expression matching. We therefore extend the semantics by using the tree monad instead of the list monad, where the size of the tree abstracts the number of execution steps required for matching. The tree monad used corresponds to the SearchTree monad [12] of Hackage 2, which is a Haskell library. The tree monad is defined as follows.

$$\begin{array}{rcl} \alpha\ \mathit{tree} & ::= & \mathrm{Unit}\ \alpha \\ & | & \mathrm{Fail} \\ & | & \mathrm{Or}\ (\alpha\ \mathit{tree}) \times (\alpha\ \mathit{tree}) \end{array}$$

The bind function $\gg\!=$ is defined as follows.

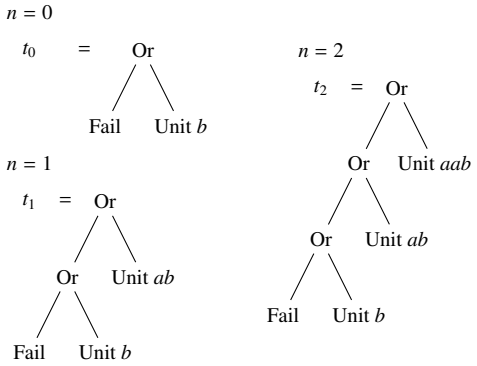


Fig. 1 Trees of $T[\![a^*]\!]a^nb$

$$
\begin{aligned}
\gg\!=_{\text{tree}} &:: \alpha\ \text{tree} \to (\alpha \to \beta\ \text{tree}) \to \beta\ \text{tree} \\
\text{Unit}\ w \gg\!=_{\text{tree}} f &= f\ w \\
\text{Fail} \gg\!=_{\text{tree}} f &= \text{Fail} \\
\text{Or}(t_1, t_2) \gg\!=_{\text{tree}} f &= \text{Or}(t_1 \gg\!=_{\text{tree}} f, t_2 \gg\!=_{\text{tree}} f)
\end{aligned}
$$

The semantics of regular expression matching is then extended by the tree monad so that it abstracts its execution steps.

Definition 3.2 (Tree-based Semantics of Regular Expression Matching)**.**

$$
\begin{aligned}
T[\![r]\!] &:: \Sigma^* \to \Sigma^*\ tree \\
T[\![\epsilon]\!]w &= \text{Unit}\ w \\
T[\![c]\!]w &= \begin{cases} \text{Unit}\ w' & \text{if } w = cw' \\ \text{Fail} & \text{otherwise} \end{cases} \\
T[\![r_1 r_2]\!]w &= T[\![r_1]\!]w \gg\!= \lambda w'.T[\![r_2]\!]w' \\
T[\![r_1|r_2]\!]w &= \text{Or}(T[\![r_1]\!]w, T[\![r_2]\!]w) \\
T[\![r_1^*]\!]w &= \text{Or}(T[\![r_1]\!]w \gg\!= \lambda w'.T[\![r_1^*]\!]w', \text{Unit}\ w)
\end{aligned}
$$

The function $T[\![r]\!]$ preserves the priority of regular expression matching. For example, t_1 is a higher priority than t_2 for Or(t_1,t_2). We then have the following theorem.

Theorem 3.3.

$$\text{leaflist}(T[\![r]\!]w) = N[\![r]\!]w$$

where the function leaflist *is defined as follows.*

$$
\begin{aligned}
&\text{leaflist (Unit}\ w) &&= [w] \\
&\text{leaflist Fail} &&= [] \\
&\text{leaflist (Or}(t_1, t_2)) &&= (\text{leaflist}\ t_1) {+\!+} (\text{leaflist}\ t_2)
\end{aligned}
$$

The size of the tree returned by the function $T[\![r]\!]$ is proportional to the time required to search all possible results of regular expression matching. We can therefore discuss the running time for searching all possible results of matching w with r by deciding the size of the tree $T[\![r]\!]w$.

Example 3.4 (Case of $O(n)$)**.** *Let us consider* $t_n = T[\![a^*]\!]a^nb$*. Figure* 1 *shows* t_0, t_1*, and* t_2*. The size of each tree is* $\text{size}(t_0) = 3$*,* $\text{size}(t_1) = 5$*, and* $\text{size}(t_2) = 7$*. Then,* $\text{size}(t_n) = 2n + 3$*. The actual execution times by* PCRE *are* 31.0ms *for* $n = 1,000,000$*,* 61.9ms *for* $n = 2,000,000$*, and* 124ms *for* $n = 4,000,000$ [*2]*.*

Example 3.5 (Case of $O(n^2)$)**.** *Let us consider* $u_n = T[\![a^*a^*]\!]a^nb$*. Figure* 2 *shows* u_0, u_1*, and* u_n*. The size of each tree is as follows.*

[*2] We show the average time for five executions in the C language.

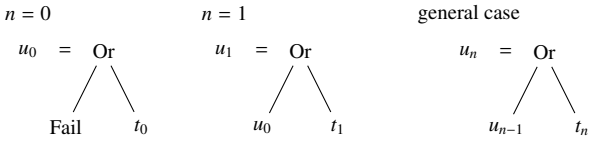


Fig. 2 Trees of $T[\![a^*a^*]\!]a^nb$

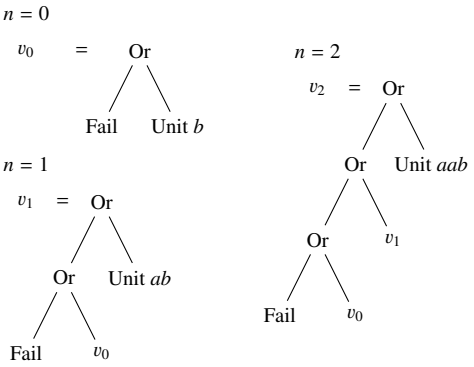


Fig. 3 Trees of $T[\![(aa^*)^*]\!]a^nb$

$$
\begin{aligned}
\text{size}(u_0) &= 2 + \text{size}(t_0) = 5 \\
\text{size}(u_1) &= \text{size}(u_0) + \text{size}(t_1) + 1 = 11 \\
\text{size}(u_n) &= \text{size}(u_{n-1}) + \text{size}(t_n) + 1
\end{aligned}
$$

$\text{size}(u_n)$ *is expanded as follows.*

$$
\begin{aligned}
\text{size}(u_n) &= \sum_{k=1}^{n} (\text{size}(t_k) + 1) + \text{size}(t_0) + 2 \\
&= \sum_{k=1}^{n} (2n + 4) + 5 \\
&= n^2 + 5n + 5
\end{aligned}
$$

The actual execution times by PCRE *are* 1.41s *for* $n = 10,000$*,* 5.70s *for* $n = 20,000$*, and* 22.7s *for* $n = 40,000$*.*

Example 3.6 (Case of $O(2^n)$)**.** *Let us consider* $v_n = T[\![(aa^*)^*]\!]a^nb$*. Figure* 3 *shows* v_0, v_1*, and* v_2*. The size of each tree is as follows.*

$$
\begin{aligned}
\text{size}(v_0) &= 3 \\
\text{size}(v_1) &= \text{size}(v_0) + 4 = 7 \\
\text{size}(v_2) &= \text{size}(v_0) + \text{size}(v_1) + 5 = 15
\end{aligned}
$$

The size of v_n *is calculated as follows.*

$$
\begin{aligned}
\text{size}(v_n) &= \sum_{k=1}^{n-1} \text{size}(v_k) + n + 3 \\
&= \text{size}(v_{n-1}) + \sum_{k=1}^{n-2} \text{size}(v_k) + (n-1) + 3 + 1 \\
&= 2\text{size}(v_{n-1}) + 1 \\
&= \sum_{k=0}^{n+1} 2^k \\
&= 2^{n+2} - 1
\end{aligned}
$$

The actual execution times by PCRE *are* 1.07s *for* $n = 23$*,* 2.14s *for* $n = 24$*, and* 4.27s *for* $n = 25$*. If the limit on execution steps is set to* $1,000,000,000$*, then the limit is exceeded for* $n = 28$*.*

The function $T[\![r]\!]$ searches all results matching an input string

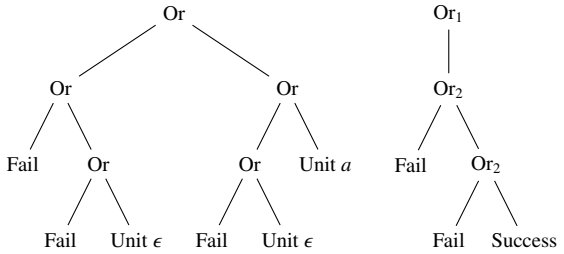


Fig. 4 Trees of $T[\![a^*a^*]\!]a$ and first$(T[\![a^*a^*]\!]a)$

with the regular expression r. However, in actual implementations based on backtracking, once a matching succeeds, the execution of matching stops and no other results are searched. Therefore, we need to construct a tree that represents the execution until the first success. To represent such trees, we revise the definition of trees as follows.

$$\begin{array}{rcl} \mathit{ctree} & ::= & \text{Success} \\ & | & \text{Fail} \\ & | & \text{Or}_1\ \mathit{ctree} \\ & | & \text{Or}_2\ \mathit{ctree} \times \mathit{ctree} \end{array}$$

In Σ^* *tree*, Unit ϵ represents a successful matching since it consumes the whole input string. From this consideration, we introduce a function that transforms Σ^* *tree* into *ctree*.

We first introduce the following function that checks whether a subtree contains a successful match.

$$\begin{array}{ll} \text{have\_succ (Unit } w) & = \begin{cases} \text{true} & \text{if } w = \epsilon \\ \text{false} & \text{otherwise} \end{cases} \\ \text{have\_succ Fail} & = \text{false} \\ \text{have\_succ (Or}(t_1, t_2)) & = \text{have\_succ } t_1 \vee \text{have\_succ } t_2 \end{array}$$

Then, the following function transforms Σ^* *tree* into *ctree*.

$$\begin{array}{ll} \text{first (Unit } w) & = \begin{cases} \text{Success} & \text{if } w = \epsilon \\ \text{Fail} & \text{otherwise} \end{cases} \\ \text{first Fail} & = \text{Fail} \\ \text{first (Or}(t_1, t_2)) & = \begin{cases} \text{Or}_1\ (\text{first } t_1) \\ \quad \text{if have\_succ } t_1 = \text{true} \\ \text{Or}_2(\text{first } t_1, \text{first } t_2) \\ \qquad\qquad \text{otherwise} \end{cases} \end{array}$$

For matching a string w with a regular expression r, first$(T[\![r]\!]w)$ is the tree representing the execution steps until the first successful match.

Example 3.7. *Let us consider* $T[\![a^*a^*]\!]a$. *Figure* 4 *shows* $T[\![a^*a^*]\!]a$ *and* first$(T[\![a^*a^*]\!]a)$. *At the root of* $T[\![a^*a^*]\!]a$, Or *is transformed to* Or_1 *because its left subtree contains* Unit ϵ. *Because other nodes do not contain* Unit ϵ *in their left subtrees, the structure of* $T[\![a^*a^*]\!]a$ *is preserved by transforming* Or *into* Or_2.

4. Tree Transducers of Linear Size Increase

4.1 Tree Automata

A tree automaton is an extension of an automaton that works on trees instead of strings [2]. A top-down tree automaton starts its computation from the root of a tree and moves on to its leaves, whereas a bottom-up tree automaton starts its computation from the leaves of a tree and moves on to its root. In this paper, we mainly use bottom-up tree automata.

Definition 4.1 (Bottom-up tree automata)**.** *A bottom-up tree automaton is a tuple* (P, Σ, h) *where* P *is a finite set of states,* Σ *is a ranked alphabet, and* h *is a family of functions called a transition function. For* $\sigma^{(k)}$, h_σ *is a function of type* $P^k \to P$.

For better readability, if we have $h_\sigma(p_1, \ldots, p_k) = p$, we write this transition as follows.

$$\sigma(p_1, \ldots, p_k) \to p$$

The transition function h is extended to $\tilde{h}$ over T_Σ as follows.

$$\tilde{h}(\sigma(t_1, \ldots, t_k)) = h_\sigma(\tilde{h}(t_1), \ldots, \tilde{h}(t_k))$$

In this paper, we sometimes use a bottom-up tree automaton (P, Σ, h, Q_f) with a set of final states Q_f. We then say that t is accepted by the tree automaton if $\tilde{h}(t) \in Q_f$. A tree language is *regular* if it is accepted by some tree automaton.

Example 4.2 (Tree automata)**.** *Let us consider a tree automaton over a Boolean expression consisting of* 0, 1, *and disjunction. The automaton is described by* (P, Σ, h, Q_f), *where* $\Sigma = \{\text{or}^{(2)}, 0^{(0)}, 1^{(0)}\}$, $P = \{p_0, p_1\}$, *and* h *consists of the following rules.*

$$\begin{array}{ll} 0 \to p_0, & 1 \to p_1 \\ \text{or}(p_1, p_0) \to p_1, & \text{or}(p_0, p_1) \to p_1 \\ \text{or}(p_1, p_1) \to p_1, & \text{or}(p_0, p_0) \to p_0 \end{array}$$

We then have the following transition for or(or(1, 0), 1).

$$\begin{array}{l} \quad \text{or}(\text{or}(1,0),1) \ \Rightarrow\ \text{or}(\text{or}(p_1,0),1) \\ \Rightarrow\ \text{or}(\text{or}(p_1,p_0),1) \ \Rightarrow\ \text{or}(p_1,1) \\ \Rightarrow\ \text{or}(p_1,p_1) \ \Rightarrow\ p_1 \end{array}$$

4.2 Tree Transducers

A tree transducer is an extension of a tree automaton that takes a tree as input and produces a tree as output. As for tree automata, there are two variants, namely top-down and bottom-up tree transducers. In this paper, we use an extension of top-down tree transducers, called top-down tree transducers with regular lookahead (or T^R for short).

We now introduce some notations for the definition of top-down tree transducers with regular lookahead. Let the set of variables $\{x_1, x_2, \ldots, x_k\}$ be denoted by X_k. We regard a tuple consisting of a state and a variable as a symbol of rank 0, and write $\langle Q, X_k\rangle = \{\langle q, x_i\rangle \mid q \in Q, x_i \in X_k\}$.

Definition 4.3 (Top-down tree transducer with regular lookahead)**.** *A top-down tree transducer with regular lookahead is a tuple* $(Q, P, \Sigma, \Delta, Q_0, R, h)$ *where* Q *is a finite set of states,* Σ *is an input ranked alphabet,* Δ *is an output ranked alphabet,* $Q_0 \subseteq Q$ *is a set of initial states,* (P, Σ, h) *is a bottom-up tree automaton, called a lookahead automaton, and* R *is a set of transition rules.*

Each transition rule in R *has the following form:*

$$\langle q, \sigma(x_1, \ldots, x_k)\rangle \to \zeta \quad \langle p_1, \ldots, p_k\rangle$$

where $\zeta \in T_{\langle Q, X_k\rangle \cup \Delta}$ *and* $\langle p_1, \ldots, p_k\rangle$ *is the condition represented by a tuple of states* $p_1, \ldots, p_k$ *of the lookahead automaton. When the tree transducer reads* σ *at state* q, *the tree is rewritten to* ζ *if* $\tilde{h}(x_i) = p_i$ *for* $i \in [k]$.

If the transition rule

$$\langle q, \sigma(x_1, \ldots, x_k)\rangle \to \zeta \quad \langle p_1, \ldots, p_k\rangle$$

applies for all $p_1, \ldots, p_k \in P$, it is written in short form as follows.

$$\langle q, \sigma(x_1, \ldots, x_k)\rangle \to \zeta$$

In the following definition of the derivation relation for tree transducers, a tuple consisting of a state and a tree is regarded as a symbol of rank 0: $\langle Q, T_\Sigma\rangle = \{\langle q, t\rangle \mid q \in Q, t \in T_\Sigma\}$. We now define the derivation relation $\Rightarrow_M$ of a top-down tree transducer M with regular lookahead.

Definition 4.4 (Derivation relation)**.** *Let $M = (Q, P, \Sigma, \Delta, Q_0, R, h)$ be a T^R. Let us consider the following transition rule of M.*

$$\langle q, \sigma(x_1, \ldots, x_k)\rangle \to \zeta \quad \langle p_1, \ldots, p_k\rangle$$

For $\xi_1, \xi_2 \in T_{\langle Q, T_\Sigma\rangle \cup \Delta}$, if the following conditions hold, then we say ξ_2 is derived by ξ_1 and write $\xi_1 \Rightarrow_M \xi_2$.
(1) $\xi_1/u = \langle q, \sigma(s_1, \ldots, s_k)\rangle$.
(2) $\tilde{h}(s_i) = p_i$ $(1 \le i \le k)$.
(3) $\xi_2 = \xi_1[u \leftarrow \zeta']$ where ζ' is the following tree.

$$\zeta' = \zeta[\langle q', x_i\rangle \leftarrow \langle q', s_i\rangle \mid \langle q', x_i\rangle \in \langle Q, X_k\rangle]$$

The translation realized by a tree transducer M is denoted by τ_M. For $t \in T_\Sigma$, $\tau_M(t) = \{t' \in T_\Delta \mid q \in Q_0.\ \langle q, t\rangle \Rightarrow_M^* t'\}$. The translation is extended for a tree language T: $\tau_M(T) = \bigcup_{t \in T} \tau_M(t)$.

Example 4.5 (Top-down tree transducer with regular lookahead)**.** *For T_Σ of Example* 4.2*, we consider a T^R that translates a node into its left subtree if the left subtree is evaluated to* 1*. The transducer is given by $(Q, P, \Sigma, \Delta, Q_0, R, h)$ where $\Delta = \Sigma$, $Q = \{q\}$, the lookahead automaton (P, Σ, h) is that of Example* 4.2*, and R has the following rules.*

$$\begin{array}{lcll}
\langle q, 0\rangle \to 0\,, & & \langle q, 1\rangle \to 1 & \\
\langle q, \mathrm{or}(x_1, x_2)\rangle & \to & \mathrm{or}(\langle q, x_1\rangle, \langle q, x_2\rangle) & \langle p_0, p_0\rangle \\
\langle q, \mathrm{or}(x_1, x_2)\rangle & \to & \mathrm{or}(\langle q, x_1\rangle, \langle q, x_2\rangle) & \langle p_0, p_1\rangle \\
\langle q, \mathrm{or}(x_1, x_2)\rangle & \to & \langle q, x_1\rangle & \langle p_1, p_0\rangle \\
\langle q, \mathrm{or}(x_1, x_2)\rangle & \to & \langle q, x_1\rangle & \langle p_1, p_1\rangle
\end{array}$$

We now show the transition of $\mathrm{or}(\mathrm{or}(0,1), \mathrm{or}(1,0))$ *by the transducer.*

$$\begin{array}{ll}
& \langle q, \mathrm{or}(\mathrm{or}(0,1), \mathrm{or}(1,0))\rangle \\
\Rightarrow & \langle q, \mathrm{or}(0,1)\rangle \qquad (1) \\
\Rightarrow & \mathrm{or}(\langle q, 0\rangle, \langle q, 1\rangle) \qquad (2) \\
\Rightarrow & \mathrm{or}(0, \langle q, 1\rangle) \qquad (3) \\
\Rightarrow & \mathrm{or}(0, 1) \qquad (4)
\end{array}$$

(1) *Because the tree has the form $\langle q, \mathrm{or}(x_1, x_2)\rangle$, the transition is determined by $\tilde{h}(\mathrm{or}(0,1)) = p_1$ and $\tilde{h}(\mathrm{or}(1,0)) = p_1$. The tree is therefore rewritten by the following rule.*

$$\langle q, \mathrm{or}(x_1, x_2)\rangle \to \langle q, x_1\rangle \qquad \langle p_1, p_1\rangle$$

(2) *Because $\tilde{h}(0) = p_0$ and $\tilde{h}(1) = p_1$, the tree is rewritten by the following rule.*

$$\langle q, \mathrm{or}(x_1, x_2)\rangle \to \mathrm{or}(\langle q, x_1\rangle, \langle q, x_2\rangle) \qquad \langle p_0, p_1\rangle$$

(3) *The tree is rewritten by the rule $\langle q, 0\rangle \to 0$.*
(4) *The tree is rewritten by the rule $\langle q, 1\rangle \to 1$.*

If the size of a tree obtained by the translation realized by a tree transducer M is linearly bounded with respect to the size of the input tree, we say that M is of *linear size increase*: there exists $c \in \mathbb{N}$ such that $\mathrm{size}(t') \le c \cdot \mathrm{size}(t)$ for any $t \in T_\Sigma$ and $t' \in \tau_M(t)$.

4.3 Finite Copying

We introduce a restriction of T^R, called *finite copying*. If a tree transducer is finite copying, then it is of linear size increase [7]. Informally, a tree transducer M is finite copying if every subtree t/u for $u \in V(t)$ is translated boundedly many times by M for any tree t. For the precise definition, we need to define the *state sequence* at a subtree t/u that is the sequence of states translating the subtree t/u. To define state sequences, we introduce the extension of a T^R that reads a state as an input symbol.

Definition 4.6 (Extension. Definition4.1 [7])**.** *Let $M = (Q, P, \Sigma, \Delta, Q_0, R, h)$ be a T^R. The extension of M, denoted by $\hat{M}$, is a T^R $(Q, P, \hat{\Sigma}, \hat{\Delta}, Q_0, \hat{R}, \hat{h})$ where $\hat{\Sigma} = \Sigma \cup \{p^{(0)} \mid p \in P\}$, $\hat{\Delta} = \Delta \cup \langle\!\langle Q, P\rangle\!\rangle$, $\hat{R} = R \cup \{\langle q, p\rangle \to \langle\!\langle q, p\rangle\!\rangle \mid \langle q, p\rangle \in \langle Q, P\rangle\}$, $\hat{h}_p() = p$ for $p \in P$, and $\hat{h}_\Sigma(p_1, \ldots, p_k) = h_\sigma(p_1, \ldots, p_k)$ for $\sigma \in \Sigma^{(k)}, p_1, \ldots, p_k \in P$.*

We then define state sequences by using Definition 4.6.

Definition 4.7 (State sequence. Definition4.4 [7])**.** *Let $M = (Q, P, \Sigma, \Delta, Q_0, R, h)$ be a T^R, $t \in T_\Sigma$ and $u \in V(t)$. Let $p = h(t/u)$, $\xi = \hat{M}(s[u \leftarrow p]) \in T_{\langle\!\langle Q, \{p\}\rangle\!\rangle \cup \Delta}$, and $\{v \in V(\xi) \mid \xi[v] \in \langle\!\langle Q, \{p\}\rangle\!\rangle\} = \{v_1, \ldots, v_n\}$ where $v_1 < \ldots < v_n$. Then, the state sequence $\mathrm{sts}_M(t, u)$ of t at u is the sequence of states $q_1 \cdots q_n$ such that $\xi[v_i] = \langle\!\langle q_i, p\rangle\!\rangle$ for $i \in [n]$.*

The restriction of finite copying is defined by using Definition 4.7.

Definition 4.8 (Finite copying. Definition4.5 [7])**.** *A T^R M is finite copying if there exists $c \in \mathbb{N}$ such that $|\mathrm{sts}_M(t, u)| \le c$ for any $t \in T_\Sigma$ and $u \in V(t)$.*

Theorem 4.9 (Lemma 4.10(i)[7])**.** *It is decidable whether or not T^R M is finite copying.*

Proof. In [7], a macro tree transducer is used to construct a state sequence represented by $T_{MON(Q)}$. However, in this paper, we construct a state sequence represented by $T_{STR(Q)}$ by using a bottom-up tree transducer. This is because we do not have to use a macro tree transducer. For the definition of bottom-up tree transducers, please refer to [5].

Let $M = (Q, P, \Sigma, \Delta, Q_0, R, h)$. The bottom-up tree transducer N that constructs a state sequence is given by $(\{r\}, \Delta \cup \langle\!\langle Q, P\rangle\!\rangle, STR(Q), \{r\}, R')$. For $\langle\!\langle q, p\rangle\!\rangle \in \langle\!\langle Q, P\rangle\!\rangle$, and $\delta_k \in \Delta^{(k)}$, R' contains the following transition rules.

$$\begin{array}{ll}
\langle\!\langle q, p\rangle\!\rangle & \to r(q) \\
\delta_0 & \to r(\epsilon_S) \\
\delta_1(r(x)) & \to r(x) \\
\delta_k(r(x_1), \ldots, r(x_k)) & \\
\qquad & \to r(\bullet(x_1, \bullet(x_2, \ldots, \bullet(x_{k-1}, x_k)\ldots)))
\end{array}$$

For $t \in T_\Sigma$, $u \in V(t)$, $\tau_N(\tau_{\hat{M}}(t[u \leftarrow h(t/u)]))$ is $\mathrm{sts}_M(t, u)$ repre-

sented by $STR(Q)$. However, the trees in $\tau_N(\tau_{\hat{M}}(t[u \leftarrow h(t/u)]))$ may not be compact and may have some redundancy. We therefore make them compact by using Lemma 4.2[4] to discuss its finiteness. Then, M is finite copying if and only if compact $\tau_N(\tau_{\hat{M}}(L))$ is finite where $L = \{t[u \leftarrow h(t/u)] \mid t \in T_\Sigma, u \in V(t)\}$. Because L is regular, it is decidable whether or not $\tau_N(\tau_{\hat{M}}(L))$ is finite by using Lemma 3.8 [7].

5. Tree Transducer Simulating Regular Expression Matching

From a regular expression, we construct a tree transducer that simulates its matching and outputs the tree introduced in Section 3 that abstracts the execution steps of regular expression matching based on backtracking. If the transducer is of linear size increase, the running time of the matching for the regular expression is determined as linear. The construction of automata and transducers in this section is based on the construction of Frisch and Cardelli [8].

5.1 Construction of Automata by Frisch and Cardelli

In preparation for the construction of the tree transducer, we review the construction of an automaton from a regular expression by Frisch and Cardelli [8].

We first define notations for list operations. We write $x :: l$ for the list obtained by adding the element x at the beginning of the list l and $l :: x$ for the list obtained by adding x at the end of l. The operation :: is extended for a set of lists: $x :: S = \{x :: l \mid l \in S\}$.

We introduce locations of a regular expression: a location is a sequence consisting of {fst, snd, lft, rgt, star}. The set of locations in a regular expression r, denoted by $\lambda(r)$, is defined as follows.

$$\begin{array}{lcl}
\lambda(\epsilon) & = & \{[]\} \\
\lambda(c) & = & \{[]\} \\
\lambda(r_1 r_2) & = & \{[]\} \cup \mathrm{fst} :: \lambda(r_1) \cup \mathrm{snd} :: \lambda(r_2) \\
\lambda(r_1|r_2) & = & \{[]\} \cup \mathrm{lft} :: \lambda(r_1) \cup \mathrm{rgt} :: \lambda(r_2) \\
\lambda(r_1^*) & = & \{[]\} \cup \mathrm{star} :: \lambda(r_1)
\end{array}$$

The subexpression of r at a location l, denoted by $r.l$, is defined as follows.

$$\begin{array}{lcl}
r.[] & = & r \\
(r_1 r_2).(\mathrm{fst} :: l) & = & r_1.l \\
(r_1 r_2).(\mathrm{snd} :: l) & = & r_2.l \\
(r_1|r_2).(\mathrm{lft} :: l) & = & r_1.l \\
(r_1|r_2).(\mathrm{rgt} :: l) & = & r_2.l \\
(r_1^*).(\mathrm{star} :: l) & = & r_1.l
\end{array}$$

After matching a string with a subexpression $r_0.l$, the rest of the string is matched with the successor of the subexpression. The successor of a subexpression is defined as follows:

$$\begin{array}{lcl}
\mathrm{succ}([]) & = & q_f \\
\mathrm{succ}(l :: \mathrm{fst}) & = & l :: \mathrm{snd} \\
\mathrm{succ}(l :: \mathrm{snd}) & = & \mathrm{succ}(l) \\
\mathrm{succ}(l :: \mathrm{lft}) & = & \mathrm{succ}(l) \\
\mathrm{succ}(l :: \mathrm{rgt}) & = & \mathrm{succ}(l) \\
\mathrm{succ}(l :: \mathrm{star}) & = & l
\end{array}$$

where q_f denotes the end of successful matching.

We now show the construction of an automaton from a regular expression, following Frisch and Cardelli [8].

Definition 5.1 (Construction of an Automata from a Regular Expression)**.** *A nondeterministic automaton $(Q, \Sigma \cup \{\epsilon\}, q_f, E)$ is constructed from a regular expression r_0 where the set of states Q is $\lambda(r_0) \cup \{q_f\}$ and the transition relation $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ contains the following rules.*

$$\begin{array}{ll}
(l, c, \mathrm{succ}(l)) \in E & \text{if } r_0.l = c \\
(l, \epsilon, \mathrm{succ}(l)) \in E & \text{if } r_0.l = \epsilon \\
(l, l :: \mathrm{fst}) \in E & \text{if } r_0.l = r_1 r_2 \\
(l, l :: \mathrm{lft}) \in E,\ (l, l :: \mathrm{rgt}) \in E & \text{if } r_0.l = r_1|r_2 \\
(l, l :: \mathrm{star}) \in E,\ (l, \mathrm{succ}(l)) \in E & \text{if } r_0.l = r_1^*
\end{array}$$

Let $L(q)$ be the set of strings accepted by the automaton of Definition 5.1, starting from an initial state $q \in Q$.

Lemma 5.2 (Lemma 2 [8])**.** *For all $l \in \lambda(r_0)$, we have $L(l) = L(r_0.l)L(\mathrm{succ(l)})$. In particular, $L([]) = L(r_0)$.*

5.2 Construction of Tree Transducers

In Section 3, we formulated $T[\![r]\!]$ that simulates regular expression matching and outputs a tree that abstracts its execution steps. In this section, we construct a transducer that outputs the tree. A string $w \in \Sigma^*$ is considered as a tree over $MON(\Sigma)$. For example, the string abc is a tree $a(b(c(\epsilon_M)))$.

Definition 5.3 (Tree Transducer Simulating Regular Expression Matching)**.** *We define a top-down tree transducer $M_{r_0} = (Q, \Sigma, \Delta, Q_0, R)$ that searches all possible matchings with regular expression r_0. The input alphabet Σ is given by $\Sigma = MON(\Sigma')$ for a set of characters Σ'. The output alphabet Δ is given by $\Delta = \{\mathrm{Success}^{(0)}, \mathrm{Fail}^{(0)}, \mathrm{Or}^{(2)}\}$. The set of states is $Q = \lambda(r_0) \cup \{q_f\}$, the set of initial states is $Q_d = \{[]\}$, and the set of transition rules R has the following rules.*

- *Case $l \in Q$.*

$$\left.\begin{array}{lcl}
\langle l, c(x)\rangle & \to & \langle \mathrm{succ}(l), x\rangle \\
\langle l, c'(x)\rangle & \to & \mathrm{Fail} \\
\langle l, \epsilon_M\rangle & \to & \mathrm{Fail}
\end{array}\right\} \quad \text{if } r_0.l = c$$

$$\begin{array}{ll}
\langle l, x\rangle \to \langle \mathrm{succ}(l), x\rangle & \text{if } r_0.l = \epsilon \\
\langle l, x\rangle \to \langle l :: \mathrm{fst}, x\rangle & \text{if } r_0.l = r_1 r_2 \\
\langle l, x\rangle \to \mathrm{Or}(\langle l :: \mathrm{lft}, x\rangle, \langle l :: \mathrm{rgt}, x\rangle) & \text{if } r_0.l = r_1|r_2 \\
\langle l, x\rangle \to \mathrm{Or}(\langle l :: \mathrm{star}, x\rangle, \langle \mathrm{succ}(l), x\rangle) & \text{if } r_0.l = r_1^*
\end{array}$$

- *Case: $q_f \in Q$.*

$$\langle q_f, \epsilon\rangle \to \mathrm{Success}$$

The top-down tree transducer M_{r_0} constructed from a regular expression r_0 reads the tree representation of an input string w and outputs the tree that abstracts the computation matching w with r_0.

It should be noted that the construction in Definition 5.3 contains ϵ-transitions and therefore does not conform to Definition 5.3 of top-down tree transducers with regular lookahead. However, the transition rules of Definition 5.3 have no loops of ϵ-transitions and can therefore be transformed easily to give a transducer without ϵ-transitions.

Example 5.4 (Tree Transducer Simulating Regular Expression Matching)**.** *We apply the top-down transducer M_r constructed from $r = ab|c^*$ to an input string ab.*

The set of states is $Q = \{[], [\mathrm{lft}], [\mathrm{lft}, \mathrm{fst}], [\mathrm{lft}, \mathrm{snd}], [\mathrm{rgt}], [\mathrm{rgt}, \mathrm{star}]\}$ *and the set of transition rules* R *contains the following rules.*

$$\begin{array}{lcl}
\langle [], x\rangle & \to & \mathrm{Or}(\langle [\mathrm{lft}], x\rangle, \langle [\mathrm{rgt}], x\rangle) \\
\langle [\mathrm{lft}], x\rangle & \to & \langle [\mathrm{lft}, \mathrm{fst}], x\rangle \\
\langle [\mathrm{lft}, \mathrm{fst}], a(x)\rangle & \to & \langle [\mathrm{lft}, \mathrm{snd}], x\rangle \\
\langle [\mathrm{lft}, \mathrm{snd}], b(x)\rangle & \to & \langle q_f, x\rangle \\
\langle [\mathrm{rgt}], x\rangle & \to & \mathrm{Or}(\langle [\mathrm{rgt}, \mathrm{star}], x\rangle, \langle q_f, x\rangle) \\
\langle [\mathrm{rgt}, \mathrm{star}], c(x)\rangle & \to & \langle [\mathrm{rgt}], x\rangle \\
\langle q_f, \epsilon_M\rangle & \to & \mathrm{Success}
\end{array}$$

where the transitions whose right hand side is Fail *are omitted.*

The following is the transition of the transducer for an input ab.

$$\begin{array}{ll}
 & \langle [], a(b(\epsilon_M))\rangle \\
\Rightarrow_{M_r} & \mathrm{Or}(\langle [\mathrm{lft}], a(b(\epsilon_M))\rangle, \langle [\mathrm{rgt}], a(b(\epsilon_M))\rangle) \\
\Rightarrow_{M_r} & \mathrm{Or}\langle [\mathrm{lft}, \mathrm{fst}], a(b(\epsilon_M))\rangle, \\
 & \quad \langle [\mathrm{rgt}], a(b(\epsilon_M))\rangle) \\
\Rightarrow_{M_r} & \mathrm{Or}(\langle [\mathrm{lft}, \mathrm{snd}], b(\epsilon_M)\rangle, \\
 & \quad \langle [\mathrm{rgt}], a(b(\epsilon_M))\rangle) \\
\Rightarrow_{M_r} & \mathrm{Or}(\langle q_f, \epsilon_M\rangle, \\
 & \quad \langle [\mathrm{rgt}], a(b(\epsilon_M))\rangle) \\
\Rightarrow_{M_r} & \mathrm{Or}(\mathrm{Success}, \\
 & \quad \langle [\mathrm{rgt}], a(b(\epsilon_M))\rangle) \\
\Rightarrow_{M_r} & \mathrm{Or}(\mathrm{Success}, \\
 & \quad \mathrm{Or}(\langle [\mathrm{rgt}, \mathrm{star}], a(b(\epsilon_M))\rangle, \langle q_f, a(b(\epsilon_M))\rangle)) \\
\Rightarrow_{M_r} & \mathrm{Or}(\mathrm{Success}, \\
 & \quad \mathrm{Or}(\mathrm{Fail}, \langle q_f, a(b(\epsilon_M))\rangle)) \\
\Rightarrow_{M_r} & \mathrm{Or}(\mathrm{Success}, \mathrm{Or}(\mathrm{Fail}, \mathrm{Fail}))
\end{array}$$

The top-down transducer constructed by Definition 5.3 outputs the tree that abstracts the execution steps searching all possible matchings. To simulate matching based on backtracking, we extend the construction in Definition 5.3 by using regular lookahead so that the output of the transition abstracts the execution that stops at the first successful matching. To simplify the presentation, we formulate the transition rules of the top-down transducer with regular lookahead in the following form. For a tree regular language L, we write

$$\langle q, \sigma(x)\rangle \xrightarrow{x \in L} \xi$$

for a transition rule that rewrites $\langle q, \sigma(x)\rangle$ into ξ when $x \in L$.

Definition 5.5 (Tree Transducer Simulating Matching Based on Backtracking)**.** *For a regular expression* r_0*, we define a top-down tree transducer* $M'_{r_0} = (Q, \Sigma, \Delta', Q_0, R')$ *with regular lookahead that simulates matching based on backtracking. The output alphabet is revised to* $\Delta' = \{\mathrm{Success}^{(0)}, \mathrm{Fail}^{(0)}, \mathrm{Or}_1{}^{(1)}, \mathrm{Or}_2{}^{(2)}\}$*. For* $r_1|r_2$ *or* r_1^**, if the branch with a higher priority succeeds, it outputs* Or_1*, containing only the subtree corresponding to that branch. If the branch with a higher priority fails, it outputs* Or_2*, containing the two subtrees corresponding to both possibilities. The transition rules of* R *are revised so that* R' *has the following rules for the cases* $r_0.l = r_1|r_2$ *and* $r_0.l = r_1^*$.

- *Case* $r_0.l = r_1|r_2$.

$$\begin{array}{ll}
\langle l, x\rangle & \xrightarrow{x \in L(l::\mathrm{lft})} \mathrm{Or}_1(\langle l :: \mathrm{lft}, x\rangle) \\
\langle l, x\rangle & \xrightarrow{x \notin L(l::\mathrm{lft})} \mathrm{Or}_2(\langle l :: \mathrm{lft}, x\rangle, \langle l :: \mathrm{rgt}, x\rangle)
\end{array}$$

- *Case* $r_0.l = r_1^*$.

$$\begin{array}{ll}
\langle l, x\rangle & \xrightarrow{x \in L(l::\mathrm{star})} \mathrm{Or}_1(\langle l :: \mathrm{star}, x\rangle) \\
\langle l, x\rangle & \xrightarrow{x \notin L(l::\mathrm{star})} \mathrm{Or}_2(\langle l :: \mathrm{star}, x\rangle, \langle \mathrm{succ}(l), x\rangle)
\end{array}$$

Example 5.6 (Tree Transducer Simulating Matching Based on Backtracking)**.** *We apply the top-down transducer* M'_r *constructed from* $r = ab|c^*$ *to an input string* ab.

The set of transition rules of M'_r *is obtained by revising the rules in Example* 5.4 *to give the following rules for the transitions from states* [] *and* [rgt].

$$\begin{array}{ll}
\langle [], x\rangle & \xrightarrow{x \in L([\mathrm{lft}])} \mathrm{Or}_1(\langle [\mathrm{lft}], x\rangle) \\
\langle [], x\rangle & \xrightarrow{x \notin L([\mathrm{lft}])} \mathrm{Or}_2(\langle [\mathrm{lft}], x\rangle, \langle [\mathrm{rgt}], x\rangle) \\
\langle [\mathrm{rgt}], x\rangle & \xrightarrow{x \in L([\mathrm{rgt}, \mathrm{star}])} \mathrm{Or}_1(\langle [\mathrm{rgt}, \mathrm{star}], x\rangle) \\
\langle [\mathrm{rgt}], x\rangle & \xrightarrow{x \notin L([\mathrm{rgt}, \mathrm{star}])} \mathrm{Or}_2(\langle [\mathrm{rgt}, \mathrm{star}], x\rangle, \langle q_f, x\rangle)
\end{array}$$

The following is the transition of the transducer for an input ab.

$$\begin{array}{ll}
 & \langle [], a(b(\epsilon_M))\rangle \\
\Rightarrow_{M'_r} & \mathrm{Or}_1(\langle [\mathrm{lft}], a(b(\epsilon_M))\rangle) \\
\Rightarrow_{M'_r} & \mathrm{Or}_1(\langle [\mathrm{lft}, \mathrm{fst}], a(b(\epsilon_M))\rangle) \\
\Rightarrow_{M'_r} & \mathrm{Or}_1(\langle [\mathrm{lft}, \mathrm{snd}], b(\epsilon_M)\rangle) \\
\Rightarrow_{M'_r} & \mathrm{Or}_1(\langle q_f, \epsilon_M\rangle) \\
\Rightarrow_{M'_r} & \mathrm{Or}_1(\mathrm{Success})
\end{array}$$

In Example 5.4*, there are leaves to the right of* Success*. On the other hand, the output of* M'_r *has no subtree to the right of* Success.

In Definition 5.5, the lookahead automaton was not explicitly given, to simplify the definition. In our implementation, we construct the lookahead automaton by constructing a top-down tree automaton from r_0 and applying the subset construction to the bottom-up tree automaton equivalent to the top-down tree automaton. Let (P, Σ, h) be the lookahead tree automaton constructed in this manner and let us consider the following transition rule.

$$\langle l, x\rangle \xrightarrow{x \in L(l::\mathrm{star})} \mathrm{Or}_1(\langle l :: \mathrm{star}, x\rangle)$$

We then have the following transition rules for $p \in P$ such that $l :: \mathrm{star} \in p$.

$$\langle l, x\rangle \to \mathrm{Or}_1(\langle l :: \mathrm{star}, x\rangle)\ \langle p\rangle$$

The top-down tree transducer with regular lookahead constructed from a regular expression r in Definition 5.5 simulates regular expression matching based on backtracking and outputs a tree that abstracts its execution steps. By then checking whether M'_r is of linear size increase, it can be decided whether the running time of the matching is linear with respect to the length of the input string. Engelfriet and Maneth showed that it is decidable to check whether a *total deterministic* transducer is finite

copying [7]. Although the construction of the transducer in this section contains ϵ-transitions, there is exactly one possible transition for all states because of lookahead. This makes the transducer total and deterministic. We can therefore apply Theorem 4.9.

6. Implementation and Experimental Results

6.1 Implementation

Based on Sections 4 and 5, we implemented an analyzer of regular expressions in OCaml. It consists of about 6,000 lines of code including empty lines and comments. It uses the parser of Sakuma et al. [15], and supports the syntax of Perl-compatible regular expressions, including the following extensions of regular expressions.

r	::=	$\cdots$	
	$\mid$	$[c_1 c_2 \ldots c_n]$	(character class)
	$\mid$	$r^?$	(optional)
	$\mid$	$r^{*?}$	(lazy repetition)
	$\mid$	r^+	(one or more repetitions)
	$\mid$	$r\{n\}\{m\}$	(repetition between n and m)

Lazy repetition $r^{*?}$ matches a repetition of r for which a shorter matching is given a higher priority. It is expanded as $r^{*?} = \epsilon | r r^{*?}$. Other extensions such as lookahead, atomic grouping, and backreference are not supported by our implementation. In our analyzer, a regular expression r not starting with ^, which matches the start of a line, is treated as $.^{*?}r$ and a regular expression r not ending with \$, which matches the end of a line, is treated as $r.^*$.

If the top-down tree transducer with regular lookahead constructed from a regular expression r is of linear size increase, the running time of matching for r is linear with respect to the length of the input string. Furthermore, if a tree transducer is finite copying, then the transducer is of linear size increase. It is therefore sound to check the time linearity of regular expression matching by checking whether the transducer is finite copying.

For top-down tree transducers with regular lookahead, finite copying is a sufficient condition of linear size increase, but not a necessary condition, in general. To check whether a transducer is of linear size increase, Engelfriet and Maneth [7] introduced a subclass of transducers called *input proper*. A transducer is input proper if any state of the transducer produces infinitely many outputs. They showed that, for this subclass, linear size increase implies finite copying and any transducer can be translated to an equivalent input proper transducer. These give a decision procedure for linear size increase.

Because our implementation does not include the translation and only checks finite copying, it is not a sound and complete checker. However, a transducer constructed from a regular expression must output either Or_1 or Or_2 when it reads a bounded number of characters. Under these conditions, we believe that linear size increase implies finite copying and therefore that our implementation is sound and complete without the translation. A formal proof of this will be addressed in our future work.

In the following, we show how to check the time linearity of regular expression matching in our implementation into more details. For the definitions of various subclasses of tree transducers, please refer to [5], [6].

Table 1 Subclasses of Tree Transducers

Subclass	Description
$LB{-}FST$	Linear bottom-up tree transducers
$T^R{-}FST$	Top-down tree transducers with regular lookahead
$QREL$	Finite state relabeling
$LHOM$	Linear homomorphism [16]

(1) The top-down tree transducer M'_r with regular lookahead is constructed from a regular expression r using Definition 5.5.

(2) If the state sequences obtained from its extension $\hat{M}'_r$ are finite, then M'_r is finite copying.

We compose the two bottom-up tree transducers N of Theorem 4.9 and C below. By applying N, we obtain the state sequences from $\tau_{M'_r}(w)$ for an input string w. The transducer C normalizes trees on $STR(\Sigma)$ to compact trees. It is given by $(\{q_\epsilon, q\}, STR(Q), STR(Q), \{q_\epsilon, q\}, R_C)$ where Q is the set of states of M'_r and R_C has the following transition rules.

$$\begin{aligned} \epsilon_S &\to q_\epsilon(\epsilon_S) \\ \sigma &\to q(\sigma) \qquad \sigma \in Q \end{aligned}$$

$$\begin{aligned} \bullet(q_\epsilon(x_1), q_\epsilon(x_2)) &\to q_\epsilon(\epsilon_S) \\ \bullet(q_\epsilon(x_1), q(x_2)) &\to q(x_2) \\ \bullet(q(x_1), q_\epsilon(x_2)) &\to q(x_1) \\ \bullet(q(x_1), q(x_2)) &\to q(\bullet(x_1, x_2)) \end{aligned}$$

For the class of linear bottom-up tree transducers $LB{-}FST$, $LB{-}FST \circ LB{-}FST \subseteq LB{-}FST$ [5]. $N \circ C$ can therefore be constructed as a linear bottom-up tree transducer.

If $\mathrm{range}(\tau_{N\circ C} \circ \tau_{\hat{M}'_r})$ is finite, then $\hat{M}'_r$ is finite copying. To check the finiteness of $\mathrm{range}(\tau_{N\circ C} \circ \tau_{\hat{M}'_r})$, we compose $N \circ C$ with a bottom-up transducer P that nondeterministically outputs a path from the root to some leaf for a tree over $STR(Q)$. P is given by $(\{q\}, STR(Q), MON(STR(Q)), \{q\}, R_P)$ where R_P has the following transition rules.

$$\begin{aligned} \sigma &\to q(\sigma(\epsilon_M)) \quad \sigma \in STR(Q)^{(0)} \\ \bullet(q(x_1), q(x_2)) &\to q(\sigma(x_1)) \\ \bullet(q(x_1), q(x_2)) &\to q(\sigma(x_2)) \end{aligned}$$

Since P is also linear, $N \circ C \circ P$ can be constructed as a linear bottom-up transducer. We constructed this composed transducer manually.

(3) For the class $T^R{-}FST$ of top-down tree transducers with regular lookahead, $T^R{-}FST \circ LB{-}FST \subseteq T^R{-}FST$ [6]. With this result, we can construct $\hat{M}'_r \circ N \circ C \circ P$ as $T^R{-}FST$ as follows.

(a) By using $LB{-}FST \subseteq QREL \circ LHOM$ [5], we decompose $N \circ C \circ P$ into two transducers M_{qrel} and M_{lhom} such that $M_{qrel} \circ M_{lhom} = N \circ C \circ P$.

(b) By using $T^R{-}FST \circ QREL \subseteq T^R{-}FST$ [6], we construct $\hat{M}'_r \circ M_{qrel}$ as $T^R{-}FST$.

(c) By using $T^R{-}FST \circ LHOM \subseteq T^R{-}FST$ [6], we construct $\hat{M}'_r \circ M_{qrel} \circ M_{lhom}$ as $T^R{-}FST$.

(4) We check whether $\mathrm{range}(\tau_{M'_r})$ is finite as follows. $\hat{M}'_r \circ M_{qrel} \circ M_{lhom}$ is a tree transducer from and to trees representing strings. It can therefore be converted into a transducer T with regular lookahead over strings. It is then converted to a

string transducer without lookahead by the subset construction.

(5) We construct an automaton A whose language is range($\tau_{\hat{M}'_r \circ M_{qrel} \circ M_{lhom}}$). If T has a transition from q to q' that consumes input w and produces output v, then A has a corresponding transition from q to q' that consumes v. The language of A is infinite if and only if there is a loop that is reachable from the initial states and to some final state. We therefore check the finiteness of range($\tau_{\hat{M}'_r \circ M_{qrel} \circ M_{lhom}}$) by removing states that are not reachable from the initial states or to any final state and by checking for the existence of a loop. If it is finite, M'_r is finite copying and the running time of matching for the regular expression r is therefore determined to be linear with respect to the length of the input.

6.2 Experimental Results

We conducted experiments that checked the time linearity of matching for regular expressions used in several existing programs. We applied our implementation to 393 regular expressions used in five programs: PHP-Fusion, phpMyAdmin, SquirrelMail, TorrentFlux, and XOOPS. We configured our implementation so that it stops and reports a time-out if it cannot decide the linearity within 900 seconds. Our implementation showed that 47 regular expressions had nonlinear running times and that 7 regular expressions caused a time-out. Results for several regular expressions are shown in Table 2. It should be noted that the running time for a regular expression that is decided as nonlinear might actually be linear for real implementations of regular matching because various optimizations might be applied. For example, although the tree corresponding to matching a^n with $(aa^*)^*b$ has the structure shown in Example 3.6 and its size is exponential in n, its running time on Perl is linear.

In Table 2, r_1 matches a string that consecutively has `<!DOCTYPE`, any number of non-alphanumerical characters, and `XHTML` or `HTML`. Since `\W` is a character class that matches non-alphanumerical characters, `\W*` does not match `<!DOCTYPE`. The running time of r_1 is therefore linear.

r_2 matches a string that is partitioned into six arbitrary substrings separated by `/`. After `.*` matches the whole input string, `/` is located by backtracking. Because `/` is located in linear time and the matching of r_2 repeats it, its running time is linear. If we were to remove `^`, then a subexpression `.*?.*` would implicitly appear. Its running time would be nonlinear.

r_3 matches a substring that starts with `</applet` or `</link`, then has an arbitrary number of characters except `>`, and ends with `>`. The matching of this regular expression has a nonlinear running time for a string of the form `</link</link`...`</link`. We have checked the running times using PCRE for strings that contain n repetitions of `</link`. The running times were 0.26 s for $n = 10,000$, 1.0 s for $n = 20,000$, and 4.1 s for $n = 40,000$. This confirms the nonlinearity of the running time.

r_4 matches a substring comprising zero or one `$` character, then two or more uppercase alphabetical characters, then zero or one `$` character, and two or more numerical characters. The running time of this regular expression is nonlinear for strings comprising uppercase alphabetical characters. If we ignore `[$]?`, the expression starts with `.*?[A-Z]+`. The matching of this part then causes nonlinear running time when the matching fails. We have checked the running times using PCRE for strings that contain n repetitions of `A`. The running times were 0.61 s for $n = 5,000$, 3.3 s for $n = 10,000$, and 14 s for $n = 20,000$. This confirms that the running time is nonlinear.

r_5 matches a string that contains `BAD` or `NO`. Because it does not start with `^`, it is treated as starting with `.*?.*`. It therefore has nonlinear running time for strings containing neither `BAD` nor `NO`. We have checked the running times for n repetitions of `A`. Using PCRE, we obtained 0.23 s for $n = 2,000,000$, 0.46 s for $n = 4,000,000$s, and 0.92 s for $n = 8,000,000$. Using Python, we obtained 1.9 s for $n = 10,000$, 7.5 s for $n = 20,000$, and 30 s for $n = 40,000$. These results can be interpreted in terms of PCRE applying some optimization to `.*?.*`.

r_6 matches a substring that starts with `content-transfer-encoding:`, then has an arbitrary number of space characters, and ends with a string comprising alphabetical characters and at most one `-`. This regular expression contains many subexpressions of the form r^* and the matching was conducted with an option that makes it case-insensitive. The tree transducer with regular lookahead constructed from r_6 had 82 states and its lookahead automaton had 63 states. The number of the states of a lookahead automaton has an exponential effect on the time to decide whether the transducer is finite copying. This causes a time-out by exceeding the limit of 900 s.

7. Related Work

Kirrage et al. developed a method to check whether the running time of a regular expression matching is exponential [13]. They introduced a nondeterministic abstract machine that simulates regular expression matching. If there is a string that has two different sequences of matching steps for this abstract machine, it is reported that the running time for the regular expression is exponential. However, this method does not consider priorities in matching, being based on the operational semantics that nondeterministically searches all the possibilities for matching. This causes false-positives for regular expression matching based on backtracking. For example, the running time for `(.*a.*|a)*` is reported as 'exponential' in their method even if it is actually linear. In constrast, the present paper is based on the semantics that precisely models priorities in regular expression matching and does not cause such false-positives. For `(.*a.*|a)*`, it decides that it has a linear running time because any string containing more than one `a` is matched with `.*a.*`, causing the matching of `a` on the right-hand side of `|` to always fail.

Pioneering work on the size of the output of a tree transducer was done by Aho and Ullman [1]. They studied the size increase caused by a generalized syntax-directed translation, which is a variant of a top-down tree transducer, and showed that it is decidable whether the size increase by the translation is $O(n^i)$ for a nonnegative integer i, and whether it is exponential. The results of Engelfriet and Maneth [7] extend these results to macro tree transducers for linear size increase. The tree transducer constructed from a regular expression in this paper intrinsically re-

Table 2 Experimental Results

Regular expression	Result	Running time
r_1 =<!DOCTYPE\W*X?HTML	linear	15.8 s
r_2 =^.*\/.*\/.*\/.*\/.*\/.*$	linear	757 s
r_3 =</*(applet|link|style|script|iframe|frame|frameset)[^>]*>	nonlinear	373 s
r_4 =([$]?[A-Z]+)([$]?\d+)	nonlinear	1.30 s
r_5 =(.*)(BAD|NO)(.*)$	nonlinear	0.970 s
r_6 =^.*(content-transfer-encoding:)\s*(\w+-?(\w+)?).*	time-out	900 s

quires lookahead and it is therefore not possible to apply the results of Aho and Ullman to the transducer. However, we believe that it will be possible to extend their method to transducers with regular lookahead and to check their other properties related to size increase.

For implementations of regular expression matching based on the theory of automata, it has been considered difficult to implement submatching that properly respects priorities in matching. Recently, several methods that respect Perl-compatible priorities have been proposed [3], [8], [9]. In particular, the regular expression library RE2 [11], which is based on automata, has attracted much attention. However, we do not consider that all the issues of regular expression matching have been solved in such implementations. For example, regular expressions in programming languages include backreference that cannot be handled by the theory of automata. In addition, it is common to use a fixed number of repetitions, defined as follows.

$$\begin{array}{rcll} r & ::= & \cdots & \\ & | & r^n & \text{(fixed number of repetition)} \end{array}$$

An implementation based on automata such as RE2 constructs an automaton by unfolding the above repetition. It may therefore construct an automaton whose size is exponential with respect to the size of a regular expression. This blowup cannot be avoided even if the implementation is based on NFA.

8. Conclusion

For regular expression matching based on backtracking, we have developed a method to check whether the matching of a given regular expression can be performed in linear time with respect to the length of the input string. We construct a top-down tree transducer with regular lookahead from a regular expression. For a given input string, this transducer outputs the tree that represents the execution steps matching the string with the regular expression. It is then possible to check the time linearity of a regular expression by checking whether or not the constructed transducer is of linear size increase.

A particular contribution of this paper is that we apply a theoretical result for tree transducers on linear size increase to a practical problem on regular expression matching. This problem is of great interest in practice because the matching of a regular expression with nonlinear running time may cause DoS vulnerabilities or pervert the behavior of matching for implementations with a limit on the number of execution steps.

Three issues should be addressed in future work. The first issue is to prove that linear size increase implies finite copying for a transducer constructed from a regular expression. We believe that this holds because the transducer must output some symbol when it reads boundedly many characters. The second issue is to generate example inputs for a regular expression with nonlinear running time. We would like to generate an example input that shows nonlinear behavior by the regular expression. The third issue involves about the construction of a tree transducer for a regular expression that includes r_1^* where $\epsilon \in L(r_1)$. The naive application of the method in this paper to such a regular expression causes nontermination by our decision procedure.

References

[1] Aho, A. V. and Ullman, J. D.: Translations on a Context-free Grammar, *Inform. and Control*, Vol. 19, pp. 439–475 (1971).

[2] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S. and Tommasi, M.: Tree Automata Techniques and Applications, Available on: `http://www.grappa.univ-lille3.fr/tata` (2007). release October, 12th 2007.

[3] Cox, R.: Implementing Regular Expressions (2007). `http://swtch.com/~rsc/regexp/`.

[4] Drewes, F. and Engelfriet, J.: Decidability of the Finiteness of Ranges of Tree Transductions, *Inform. and Comput.*, Vol. 145, pp. 1–50 (1998).

[5] Engelfriet, J.: Bottom-up and Top-down Tree Transformations - a Comparison, *Math. Syst. Theory*, Vol. 9, pp. 198–231 (1975).

[6] Engelfriet, J.: Top-down Tree Transducers with Regular Look-ahead, *Math. Syst. Theory*, Vol. 10, pp. 289–303 (1977).

[7] Engelfriet, J. and Maneth, S.: Macro Tree Translations of Linear Size Increase are MSO Definable, *SIAM J. Comput*, Vol. 32, pp. 950–1006 (2003).

[8] Frisch, A. and Cardelli, L.: Greedy Regular Expression Matching, *ICALP 2004*, pp. 618–629 (2004). LNCS 3142.

[9] Haber, S., Horne, W., Manadhata, P., Mowbray, M. and Rao, P.: Efficient Submatch Extraction for Practical Regular Expressions, *LATA*, pp. 323–343 (2013). LNCS 7810.

[10] Hazel, P.: PCRE Man Page. `http://www.pcre.org/pcre.txt`.

[11] Hosting, G. P.: re2: An Efficient, Principled Regular Expression Library. `http://code.google.com/p/re2/`.

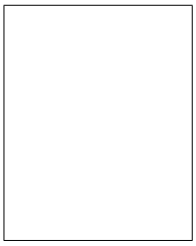
[12] IHG: Control.Monad.SearchTree. `http://hackage.haskell.org/package/tree-monad-0.3/docs/Control-Monad-SearchTree.html#t:SearchTree`.

[13] Kirrage, J., Rathnayake, A. and Thielecke, H.: Static Analysis for Regular Expression Denial-of-Service Attacks, *International Conference on Network and System Security (NSS 2013)*, pp. 135–148 (2013). LNCS 7873.

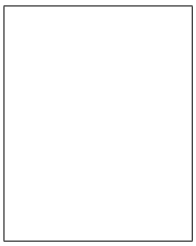
[14] OWASP: Regular Expression Denial of Service - ReDoS (2012). `https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS`.

[15] Sakuma, Y., Minamide, Y. and Voronkov, A.: Translating Regular Expression Matching into Transducers, *Journal of Applied Logic*, Vol. 10, No. 1, pp. 32–41 (2012).

[16] Thatcher, J. W.: Generalized^2 Sequential Machine Maps, *J. Comp. Syst. Sci.*, Vol. 4, pp. 339–367 (1970).



Satoshi Sugiyama received his B.Sc. degree in information science from University of Tsukuba in 2012. He is currently on the master course at University of Tsukuba. His research interests include software verification and the theory of automata and formal languages.



Yasuhiko Minamide is an Associate Professor in the Department of Computer Science, University of Tsukuba. He received his M.Sc. and Ph.D. from Kyoto University in 1993 and 1997. His research interests include the theory of programming languages and software verification. He is a member of ACM, IPSJ, and JSSST.