Technical Report 2013-002

Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions

Satoshi Okui[1] and Taro Suzuki[2]

June 5, 2013

[1]Depertment of Computer Science
Chubu University
Matsumoto 1200, Kasugai City
Aichi, 487-8501 Japan

[2]Department of Computer Science
and Engineering
The University of Aizu
Tsuruga, Ikki-Machi,
Aizu-Wakamatsu City
Fukushima, 965-8580 Japan

Technical Report 2013-002

Title:
Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions

Authors:
Satoshi Okui and Taro Suzuki

Key Words and Phrases:
regular expression, automata, pattern matching

Abstract:
This paper offers a new efficient regular expression matching algorithm which follows the leftmost-longest rule specified in POSIX 1003.2 standard. The algorithm basically emulates the subset construction without backtracking, so that its computational cost even in the worst case does not explode exponentially; the time complexity of the algorithm is $O(m(n^2+c))$, where m is the length of a given input string, n the number of occurrences of the most frequently used letters in a given regular expression and c the number of subexpressions to be used for capturing substrings plus the number of repetition operators. The correctness of the algorithm is given with respect to a formalization of the leftmost-longest semantics by means of a priority order on parse trees.

Report Date:
6/5/2013

Written Language:
English

Any Other Identifying Information of this Report:

Distribution Statement:
First Issue: 5 copies

Supplementary Notes:

Language Processing Systems Laboratory
The University of Aizu
Aizu-Wakamatsu
Fukushima 965-8580
Japan

Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions *

Satoshi Okui
Department of Computer Science, Chubu University
okui@cs.chubu.ac.jp

Taro Suzuki
School of Computer Science and Engineering, The University of Aizu
taro@u-aizu.ac.jp

Abstract

This paper offers a new efficient regular expression matching algorithm which follows the leftmost-longest rule specified in POSIX 1003.2 standard. The algorithm basically emulates the subset construction without backtracking, so that its computational cost even in the worst case does not explode exponentially; the time complexity of the algorithm is $O(m(n^2 + c))$, where m is the length of a given input string, n the number of occurrences of the most frequently used letters in a given regular expression and c the number of subexpressions to be used for capturing substrings plus the number of repetition operators. The correctness of the algorithm is given with respect to a formalization of the leftmost-longest semantics by means of a priority order on parse trees.

*This work is supported by Japan Society for Promotion of Science, Basic Research (C) No.22500019.

1 Introduction

Disambiguation in regular expression matching is crucial for many applications such as string replacement in document processing. Although POSIX 1003.2 standard requires the ambiguity of regular expressions to be resolved by following the *leftmost-longest* rule, most POSIX adopting implementations actually indicate, more or less, different behaviors. This is, in part, due to the description of the leftmost-longest rule, which is informally given in natural language, not in any formal method.

For implementations that rely on backtracking [12], strictly following the leftmost-longest rule forces loss of efficiency; the greedy, or the first match, policy is easier for such systems to follow. For this problem, some recent implementations [10, 9, 3] take approaches based on automata, realising the leftmost-longest semantics very efficiently. While those implementations are notable for their production level quality, theoretical considerations supporting the correctness of those approaches have not yet given sufficiently; we can find such a fundamental study [6] for the greedy semantics though.

This paper presents a new regular expression matching algorithm which follows the leftmost-longest semantics. We focus on reliability issues so that a significant part of our study is devoted to proving the correctness of the algorithm as well as formalizing the leftmost-longest semantics. The time complexity of our algorithm is $O(m(n^2 + c))$ where m is the length of a given input string, n the number of occurrences of the most frequently used letters in a given regular expression and c the number of subexpressions to be used for *capturing* portions of the input string plus the number of repetition operators.

Our discussion for achieving that algorithm proceeds in three steps as follows. First, in Sec. 2, we formalize the leftmost-longest rule based on *parse trees*. A parse tree for a given input string represents a way of accepting that string; e.g., which alternative is selected or how many times each iteration is performed, and so forth. Imposing a priority order on parse trees results in a straightforward interpretation of the leftmost-longest rule, which serves as a basis for our later discussion. We restrict ourselves to consider *canonical* parse trees for avoiding the matches which is unacceptable in the leftmost-longest semantics. Next, Sec. 3 introduces a slight extension of traditional *position automata* [11, 7] in order to enumerate canonical parse trees. In Sec. 4, we introduce the regular expression matching algorithm, which basically performs subset construction at runtime. This algorithm incrementally compares, in each step, the priority of any two paths to eliminate unnecessary ones, so that the number of paths in each step never exceeds the number of states. The validity of this incremental pruning is stated with respect to the formal definition of the leftmost-longest semantics. Finally, Sec. 5 is devoted to a brief comparison with other studies.

2 Formalizing the leftmost-longest semantics

2.1 Basics

A *regular expression* is an expression of the form: $r ::= 1 \mid a \mid r{\cdot}r \mid r{+}r \mid r{*}$ where a is a letter in an alphabet Σ. Our definition does not include the regular expression $\mathtt{0}$ that denotes the empty language. We consider the regular expressions as abstract syntax trees (AST) and do not take care of operator precedence. The set of *parse trees* for r, written $\mathtt{PT}(r)$, is defined as follows:

$$\begin{aligned}
\mathtt{PT}(1) &= \{1\} \\
\mathtt{PT}(a) &= \{a\} \\
\mathtt{PT}(r_1{\cdot}r_2) &= \{t_1{\cdot}t_2 \mid t_i \in \mathtt{PT}(r_i) \text{ for } i = 1, 2\} \\
\mathtt{PT}(r_1{+}r_2) &= \{\mathtt{L}(t) \mid t \in \mathtt{PT}(r_1)\} \cup \{\mathtt{R}(t) \mid t \in \mathtt{PT}(r_2)\} \\
\mathtt{PT}(r{*}) &= \bigcup_{n \geq 0} \{\mathtt{I}(t_1, \ldots, t_n) \mid t_1, \ldots, t_n \in \mathtt{PT}(r)\}
\end{aligned}$$

Note that parse trees are *unranked* in the sense that an `I`-node has an arbitrary number of children (possibly none). Using unranked trees has more benefits than using ranked trees; especially, it makes a close relationship with the original regular expression much clearer. Basically, parse trees are obtained from the AST of a regular expression by horizontally expanding children of $*$-nodes. Hence, the height of parse trees for r never exceeds that of r.

Obviously, a leaf of a parse tree is either 1 or a letter in Σ. Reading those letters from left to right, we obtain a string *derived* from that parse tree. We write $\mathtt{PT}(r, w)$ for the set of canonical parse trees deriving w. We sometimes specify each character of w by 0-based index; we write $w[i]$ ($0 \leq i < |w|$) for the i-th character. Likewise, $w[i, j]$ stands for a substring $w[i] \ldots w[j-1]$ of w.

A *position* is a possibly empty sequence of natural numbers, which specify the "path" from the root node to a subtree. The root position is the empty sequence denoted as Λ. Following the standard manner, , e.g., [1], we assign a position to each node of parse trees, except that the child of the `R`-node has a position starting with 2 rather than 1. (In other words, we assume some place holder at the place of the first sibling of `R`-node.) For other kind of nodes, the first sibling always has a position starting with 1. For a parse tree t, we denote the set of positions in t as $\mathtt{Pos}(t)$ and the subterm of t at position p as $t|_p$. The root node of $t|_p$ and the number of children (immediate subtrees) of $t|_p$ are respectively denoted by $\mathtt{node}_t(p)$ and $\mathtt{arity}_t(p)$.

Likewise, we consider $\mathtt{Pos}(r)$ and $r|_p$ for the AST of a regular expression r.

We write $p_1 \lessdot p_2$ iff p_1 precedes p_2 in the lexicographic order, while $p_1 \leqslant p_2$ if p_1 is a prefix of p_2, i.e., $p_1 q = p_2$ for some possibly empty q. We also write $p_1 \prec p_2$ for $p_1 \leqslant p_2$ and $p_1 \neq p_2$.

A binary relation R on a set A is called *strict total order* if it is (1) *transitive*, i.e., aRb & bRc implies aRc for any $a, b, c \in A$, and is (2) *trichotomous*, i.e., exactly one of aRb, bRa and $a = b$ holds. Note that any *irreflexive* (i.e., aRa does not holds for any $a \in A$), trasitive binary relation R on A is a strict total order if either aRb or bRa holds for any $a, b \in A$ such that $a \neq b$.

Let t be a parse tree. The *norm* of t at $p \in \mathtt{Pos}(t)$, written $\| t \|_p$, is the number of letters in Σ that occur in $t|_p$. As a convention, we define $\| t \|_p = -1$ for p not in $\mathtt{Pos}(t)$. $\| t \|_\Lambda$ is abbreviated as $\| t \|$.

Let $s, t \in \mathtt{PT}(r, w)$. A position $p \in \mathtt{Pos}(s) \cup \mathtt{Pos}(t)$ is called a *distinct position* of s and t unless $p \in \mathtt{Pos}(s) \cap \mathtt{Pos}(t)$. For a distinct position of s and t, we have $\| s \|_p > \| t \|_p = -1$ or else $\| t \|_p > \| s \|_p = -1$. The following lemma is just the contraposition of this:

Lemma 1 *Let $s, t \in \mathtt{PT}(r, w)$ be parse trees for a regular expression r and w a string. For any $p \in \mathtt{Pos}(s) \cup \mathtt{Pos}(t)$, $\| s \|_p = \| t \|_p$ implies $p \in \mathtt{Pos}(s) \cap \mathtt{Pos}(t)$.*

The following lemma is used everywhere in this paper.

Lemma 2 *Let $s, t \in \mathtt{PT}(r, w)$ be parse trees for a regular expression r and a string w. The following statements are equivalent:*

1. $s = t$
2. $\| s \|_p = \| t \|_p$ *for any* $p \in \mathtt{Pos}(s) \cup \mathtt{Pos}(t)$
3. $\mathtt{Pos}(s) = \mathtt{Pos}(t)$

PROOF $1 \Rightarrow 2$ is obvious. $2 \Rightarrow 3$ follows by LEMMA 1. What remains is $3 \Rightarrow 1$, which is shown by structual induction on r. Suppose $\mathtt{Pos}(s) = \mathtt{Pos}(t)$. The cases $\mathtt{node}_r(\Lambda) = 1$ and $\mathtt{node}_r(\Lambda) = \cdot$ are trivial. In case $\mathtt{node}_r(\Lambda) = +$, s and t should have the same root symbol, say $\mathtt{L}$: $s = \mathtt{L}(s_1)$ and $t = \mathtt{L}(t_1)$. On the other hand, if $\mathtt{node}_r(\Lambda) = *$, the root of s and t should have the same arity: $s = \mathtt{I}(s_1, \ldots, s_n)$ and $t = \mathtt{I}(t_1, \ldots, t_n)$. In both cases, the result follows by induction hypothesis. □

2.2 Formalizing Leftmost-Longest Matching

As we have mentioned, the POSIX specification presents the leftmost-longest rule rater informally. We first rephrase it in a more formal way in order to make our discussion more accurate.

First of all, we consider the case of whole matching; i.e., when a given regular expression matches over the whole string. In this case, the POSIX specification [8] requires that "*each subpatterns shall match the longest possible string*." This leads the following definition of priority:

Definition 1 *(priority of parse trees) Let r be a regular expression, and w a string. For any $t_1, t_2 \in \mathtt{PT}(r, w)$, we say t_1* is prior to *t_2 with respect to the* decision position *$p \in \mathtt{Pos}(t_1)$, written $t_1 \lessdot_p t_2$, if the following conditions are satisfied:*

1. $\| t_1 \|_p > \| t_2 \|_p$
2. $\| t_1 \|_q = \| t_2 \|_q$ *for any position* $q \in \mathtt{Pos}(t_1) \cup \mathtt{Pos}(t_2)$ *such that* $q \lessdot p$

We write $t_1 \lessdot t_2$ if $t_1 \lessdot_p t_2$ for some p.

Recall that we have defined $\| t \|_p = -1$ for $p \notin \mathtt{Pos}(t)$, which corresponds to the requirement [8]: "*a null string shall be considered to be longer than no match at all.*"

The definition of priority is top-down. We next give another characterization allowing us to compare parse trees in rather left-to-right manner.

Definition 2 *(first dictinct position) Let r be a regular expression, w a string, and $s, t \in \mathtt{PT}(r, w)$. A distinct position p of s and t is called the* first *distinct position if it satifies $p \lessdot p'$ for any other distinct position p' of s and t.*

Proposition 1 *Let $s, t \in \mathtt{PT}(r, w)$ be parse trees for a regular expression r and a string w such that $s \lessdot_p t$ for some decision position p. The first distinct position q of s and t satisfies the following:*

1. $s|_r = t|_r$ *for any r such that $r \lessdot q$ but $r \not\leqslant q$, and*
2. $p \leqslant q$.

PROOF We show the first statement; the second then follows immediately. Consider a position r such that $r \lessdot q$ but $r \not\leqslant q$. Since r is not a distinct position, i.e., $r \in \mathtt{Pos}(s) \cap \mathtt{Pos}(t)$, there exist both $s|_r$ and $t|_r$. Consider a position $r' \in \mathtt{Pos}(s|_r)$. Since rr' is not a distinct position, either, we have $rr' \in \mathtt{Pos}(t)$, hence, $r' \in \mathtt{Pos}(t|_r)$. Nemely, we have $\mathtt{Pos}(s|_r) \subseteq \mathtt{Pos}(t|_r)$. Similaly, we have $\mathtt{Pos}(t|_r) \subseteq \mathtt{Pos}(s|_r)$. Therefore, LEMMA 2 gives $s|_r = t|_r$. □

We need one more requirement for parse trees. The POSIX specification [8] states that "*a subexpression repeated by '*' shall not match a null expression unless this is the only match for the repetition.*" For this, we introduce the notion of *canonical* parse tree, which plays a very important role in our discussion: [1]

Definition 3 *(canonical parse tree) A parse tree t is called* canonical *if any subterm of t such as $\mathtt{I}(t_1, \ldots, t_n)$, $n \geq 2$, satisfies $\| t_i \| > 0$ for any $1 \leq i \leq n$.*

[1] The interval expression (bounded repetition), if it would be introduced directly to our framework, should be distingished from the repetition; otherwise, $1\{2\}$ would match the empty string, yielding a non-canonical parse tree $\mathtt{I}(11)$.

We write $\mathtt{CPT}(r)$ for the canonical parse trees in $\mathtt{PT}(r)$. More formally, it is given as follows:

$$\begin{aligned}
\mathtt{CPT}(1) &= \{1\} \\
\mathtt{CPT}(a) &= \{a\} \\
\mathtt{CPT}(r_1{\cdot}r_2) &= \{t_1{\cdot}t_2 \mid t_i \in \mathtt{CPT}(r_i) \text{ for } i = 1,2\} \\
\mathtt{CPT}(r_1{+}r_2) &= \{\mathtt{L}(t) \mid t \in \mathtt{CPT}(r_1)\} \cup \{\mathtt{R}(t) \mid t \in \mathtt{CPT}(r_2)\} \\
\mathtt{CPT}(r*) &= \bigcup_{n\geq 0} \{\mathtt{I}(\alpha_1 \ldots \alpha_n) \mid \alpha_1, \ldots, \alpha_n \in \mathtt{CPT}(r), \\
&\qquad\qquad \text{and } \| a_1 \| > 0, \ldots, \| a_n \| > 0 \text{ if } n \geq 2 \}
\end{aligned}$$

Likewise, $\mathtt{CPT}(r, w)$ means the canonical parse trees in$\mathtt{PT}(r, w)$.

Unlike $\mathtt{PT}(r, w)$, the set $\mathtt{CPT}(r, w)$ is finite. More specifically, we have the following:

Lemma 3 *Let r be a regular expression r, w a string and t a parse tree in $\mathtt{CPT}(r, w)$. Any node of t has at most $|w| + 2$ children.*

PROOF We only have to check I-nodes. Suppose, on the contrary, an I-node at a position $p \in \mathtt{Pos}(t)$ has more than $|w| + 2$ children. Since t is canonical and $|w| + 2 \geq 2$, we have $\| t \| \geq \| t \|_p > |w| + 2$, which contradicts to $\| t \| = |w|$. □

In practice, *partial* matching, matching for a substring of a given string, is used more extensively than whole matching. On partial matching, the specification states that "*the search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found*" and that "*if the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence is matched,*" which we formalize as follows.

Given a regular expression r, a *parse configuration* for r is defined as a triple $\langle u, t, v\rangle$ where $u, v \in \Sigma^*$ and $t \in \mathtt{CPT}(r)$. We say a parse configuration $\langle u, t, v\rangle$ *derives* a string uwv if the canonical parse tree t derives the string w. The set of parse configurations for r is denoted by $\mathtt{PC}(r)$, and, likewise, the set of parse configurations deriving w by $\mathtt{PC}(r, w)$. For any parse configurations $\langle u_1, t_1, v_1\rangle$ and $\langle u_2, t_2, v_2\rangle$ in $\mathtt{PC}(r, w)$, we write $\langle u_1, t_1, v_1\rangle \lessdot \langle u_2, t_2, v_2\rangle$ if (1) $|u_1| < |u_2|$ or (2) $|u_1| = |u_2|$ and $\| t_1 \| > \| t_2 \|$ or (3) $|u_1| = |u_2|$, $\| t_1 \| = \| t_2 \|$ and $t_1 \lessdot t_2$.

Now, we have the main result of this section.

Theorem 1 $\mathtt{PC}(r, w)$ *is a finite and strict total order set for any regular expression r and a string w.*

PROOF The finiteness of $\mathtt{PC}(r, w)$ immediately follows by the finiteness of $\mathtt{CPT}(r, w)$. This is, in turn, a consequence of LEMMA 3 since the height of parse trees in $\mathtt{PC}(r, w)$ does not exceed the height of (the AST of) r.

We show that $\mathtt{PT}(r, w)$ is a strict total order set; the result then follows since the ordering $\lessdot$ on $\mathtt{PC}(r, w)$ is an lexicographic extension of that on $\mathtt{PT}(r, w)$. To show the transitivety, consider parse trees t_1, t_2, t_3 in $\mathtt{PT}(r, w)$ such that $t_1 \lessdot_{p_1} t_2$ and $t_2 \lessdot_{p_2} t_3$. Suppose $p_1 \lessdot p_2$ or $p_1 = p_2$. Then, we have $\| t_1 \|_{p_1} > \| t_2 \|_{p_1} \geq \| t_3 \|_{p_1}$. Moreover, for any position $q \in \mathtt{Pos}(t_1) \cup \mathtt{Pos}(t_3)$ such that $q \lessdot p_1$, we have $\| t_1 \|_q = \| t_2 \|_q = \| t_3 \|_q$. Hence, DEFINITION 1 gives us $t_1 \lessdot_{p_1} t_3$. Similary, for the case $p_2 \lessdot p_1$ we also have $t_1 \lessdot_{p_2} t_3$. Therefore, the transitivity has been shown. The irriflexivity is an consequence of LEMMA 2. This lemma also implies that either $t_1 \lessdot t_2$ or $t_2 \lessdot t_1$ is true for any $t_1, t_2 \in \mathtt{PC}(r, w)$ such that $t_1 \neq t_2$. Therefore, $(\mathtt{PC}(r, w), \lessdot)$ is trichotomous, hence, a strict total order set. □

Corollary 1 $\mathtt{PC}(r, w)$ *has the least (that is, the most prior) parse configuration.*

We think of the least element as representing the matching that follows the POSIX-type leftmost-longest semantics.

Note that the least element may not exist for arbitrary parse trees; e.g., in $\mathtt{PC}(1*, \varepsilon)$, we would have an infinite decreasing sequence of arbitrary parse trees:

$$\mathtt{I}() \gtrdot \mathtt{I}(1) \gtrdot \mathtt{I}(11) \gtrdot \mathtt{I}(111) \gtrdot \dots$$

2.3 Capturing submatches

A pattern matching procedure needs to report, if the mathcing succeeds, not only the result of whole matching but also the result of submatch for each subpatterns. In our formalization, this is nothing but specifying the correspodence between positions between a regular expression and the least canonical parse tree for the regular expression. The problem here is that this correspondence is one-to-many as the subtree of $*$-node in the regular expression is extended to zero or many subtrees in the parse tree. POSIX solution for this is to focus on the latest iteration. To formalize this, we consider unfolding of positions:

Definition 4 *Let r be a regular expression and $t \in \mathtt{CPT}(r)$. For p in $\mathtt{Pos}(r)$, we define*

$\texttt{unfold}(p,t)$ *in* $\texttt{Pos}(t)$ *as follows:*

$$\begin{aligned}
\texttt{unfold}(\Lambda,t) &= \Lambda \\
\texttt{unfold}(1.p,\texttt{L}(t)) &= 1.\texttt{unfold}(p,t) \\
\texttt{unfold}(2.p,\texttt{R}(t)) &= 2.\texttt{unfold}(p,t) \\
\texttt{unfold}(i.p,t_1{\cdot}t_2) &= i.\texttt{unfold}(p,t_i) \;\; i=1,2 \\
\texttt{unfold}(1.p,\texttt{I}(t_1,\ldots,t_n)) &= n.\texttt{unfold}(p,t_n) \;\; n \geq 1
\end{aligned}$$

Definition 5 *Let t be a parse tree and $p \in \texttt{Pos}(t)$. The* location $\texttt{loc}(p,t)$ *is defined as follows:*

$$\begin{aligned}
\texttt{loc}(\Lambda,t) &= 0 \\
\texttt{loc}(1.p,\texttt{L}(t)) &= \texttt{loc}(p,t) \\
\texttt{loc}(2.p,\texttt{R}(t)) &= \texttt{loc}(p,t) \\
\texttt{loc}(1.p,t_1{\cdot}t_2) &= \texttt{loc}(p,t_1) \\
\texttt{loc}(2.p,t_1{\cdot}t_2) &= \parallel t_1 \parallel + \texttt{loc}(p,t_2) \\
\texttt{loc}(i.p,\texttt{I}(t_1,\ldots,t_n)) &= \sum_{j=1}^{i-1} \parallel t_j \parallel + \texttt{loc}(p,t_i)
\end{aligned}$$

For a parse configuration $c = \langle u,t,v\rangle$ where $t \in \texttt{CPT}(r,w)$ and a position $p \in \texttt{Pos}(r)$, we say c *captures* the interval (i,j) at a position p, where $0 \leq i \leq j \leq |uw|$, if $q = \texttt{unfold}(p,t) \in \texttt{Pos}(t)$, $i = |u| + \texttt{loc}(q,t)$, and $j = i + \parallel t|_q \parallel$; otherwise, c captures nothing at the position p.

Definition 6 *(POSIX-type pattern matching problem) Let r be a regular expression and P a subset of* $\texttt{Pos}(r)$ *called* capturing *positions. A* POSIX-type pattern matching problem *is, given a string w, to find the least canonical parse configuration $\langle u,t,v\rangle$, where $t \in \texttt{CPT}(r)$, and to report intervals captured by positions in P.*

3 Enumerating Parse Trees via Position Automata

3.1 Correctly Nested Parentheses

A *parenthesis expression* means a sequence of either parentheses, each of which is indexed by a position, or a letter in Σ. For a parenthesis expression α, its norm $\| \alpha \|$ is defined as the number of letters in α.

For each parse tree t in $\mathtt{PT}(r)$, we assign, as its string representation, a parenthesis expression $\Phi(p,t)$ as follows:

$$\begin{aligned}
\Phi(p,1) &= {}_{(p}\ )_p \\
\Phi(p,a) &= {}_{(p}a\ )_p \quad \text{where } a \in \Sigma \\
\Phi(p,t_1{\cdot}t_2) &= {}_{(p}\Phi(p.1,t_1)\ \Phi(p.2,t_2)\ )_p \\
\Phi(p,\mathtt{L}(t)) &= {}_{(p}\Phi(p.1,t)\ )_p \\
\Phi(p,\mathtt{R}(t)) &= {}_{(p}\Phi(p.2,t)\ )_p \\
\Phi(p,\mathtt{I}(t_1,\ldots,t_n)) &= {}_{(p}\Phi(p.1,t_1)\ldots\Phi(p.1,t_n)\ )_p
\end{aligned}$$

Definition 7 *(correctlly nested parenthesis expression) A parenthesis expression α is* correctlly nested *if $\alpha = \Phi(p,t)$ for some regular expression r, some position p and $t \in \mathtt{CPT}(r)$.*

Obviously, a correctlly nested parenthesis expression $\Phi(p,t)$ either of the form ${}_{(p}a)_p$ or ${}_{(p}\alpha_1\ldots\alpha_n)_p$ ($n \geq 0$) and $\alpha_1,\ldots,\alpha_n$ contains indexes less than p.

Definition 8 *For a correctlly nested parenthesis expression α, the set $\mathtt{Pos}(\alpha)$ of positions of α is defined as follows:*

$$\begin{aligned}
\mathtt{Pos}({}_{(p}a)_p) &= \{\Lambda\} \\
\mathtt{Pos}({}_{(p}\ )_p) &= \{\Lambda\} \\
\mathtt{Pos}({}_{(p}\alpha)_p) &= \{\Lambda\} \cup \{i.q \mid \alpha = {}_{p.i}\alpha'\,)_{p.i} \text{ for some } \alpha',\ q \in \mathtt{Pos}(\alpha')\} \\
\mathtt{Pos}({}_{(p}\alpha_1\ldots\alpha_n)_p) &= \{\Lambda\} \cup \{i.q_i \mid q_i \in \mathtt{Pos}(\alpha_i),\ 1 \leq i \leq n\} \quad (n \geq 2)
\end{aligned}$$

Lemma 4 *Let t be a parse tree in $\mathtt{PT}(r,w)$ for a regular expression r and a string w. Then, $\mathtt{Pos}(t) = \mathtt{Pos}(\Phi(p,t))$ for any p.*

PROOF By a straightforward structual induction on r. □

Lemma 5 *For any regular expression r and position p, $\Phi(p,\_)$ is an injection from $\mathtt{PT}(r,w)$ to correctlly nested parenthesis expressions.*

PROOF Suppose $\Phi(p,s) = \Phi(p,t)$. LEMMA 4 implies $\mathtt{Pos}(s) = \mathtt{Pos}(t)$. It follows by LEMMA 2 that $s = t$. □

For a parse configuration $\langle u, t, v\rangle$ where $t \in \mathtt{PT}(r)$, we define $\Phi(\langle u, t, v\rangle) = u\ \Phi(\Lambda, t)\ v$. We denote the set $\{\Phi(p, t) \mid t \in \mathtt{CPT}(r, w)\}$ as $\mathtt{CPT}_\Phi(r, p, w)$. Likewise, $\mathtt{PC}_\Phi(r, w)$ stands for the set $\{\Phi(c) \mid c \in \mathtt{PC}(r, w)\}$. We also write

$$\mathtt{CPT}_\Phi(r, p) = \bigcup_{w \in \Sigma^*} \mathtt{CPT}_\Phi(r, p, w) \qquad \mathtt{PC}_\Phi(r) = \bigcup_{w \in \Sigma^*} \mathtt{PC}_\Phi(r, w)$$

From the definitions of Φ and $\mathtt{CPT}(r, w)$, we have the following:

Proposition 2

$$\begin{aligned}
\mathtt{CPT}_\Phi(1, p) &= \{\ (_p\,)_p\} \\
\mathtt{CPT}_\Phi(a, p) &= \{\ (_p a\,)_p\} \\
\mathtt{CPT}_\Phi(r_1{\cdot}r_2, p) &= \{\ (_p \alpha_1 \alpha_2\,)_p \mid \alpha_i \in \mathtt{CPT}_\Phi(r_i, p.i), \textit{ for } i = 1, 2\ \} \\
\mathtt{CPT}_\Phi(r_1{+}r_2, p) &= \{\ (_p \alpha_1\,)_p \mid \alpha_1 \in \mathtt{CPT}_\Phi(r_1, p.1)\} \cup \{\ (_p \alpha_2\,)_p \mid \alpha_2 \in \mathtt{CPT}_\Phi(r_2, p.2)\} \\
\mathtt{CPT}_\Phi(r*, p) &= \bigcup_{n \geq 0} \{\ (_p \alpha_1 \ldots \alpha_n\,)_p \mid \alpha_1, \ldots, \alpha_n \in \mathtt{CPT}_\Phi(r, p.1), \\
&\qquad\qquad \textit{and } \parallel a_1 \parallel > 0, \ldots, \parallel a_n \parallel > 0 \textit{ if } n \geq 2\ \}
\end{aligned}$$

Unlike the Dyck language, any expression in $\mathtt{PC}_\Phi(r)$ has a bounded nesting depth; it does not exceed the height of (the AST of) r indeed. This means that $\mathtt{PC}_\Phi(r)$ is a regular language, thereby, recognizable (or equivalently, enumerable) by an automaton. In the next section, we will construct such automata from a regular expression r, where any "path" corresponds to a prefix of a parenthesis expression in $\mathtt{PC}_\Phi(r)$. We write the prefix closure of $\mathtt{PC}_\Phi(r)$ as $\overline{\mathtt{PC}}_\Phi(r)$.

Let α be an arbitrary parenthesis expression that is not necessarily in a range of the function Φ. We denote the number of letters in α as $\parallel \alpha \parallel$. It is sometimes convenient to "packetize" parenthesis expressions, considering the occurrences of letters as separators. For any parenthesis expression α, which is always written in the form $\beta_0 a_1 \beta_1 a_2 \ldots \beta_{n-1} a_n \beta_n$ where $a_1 \ldots a_n$ are letters and $\beta_0 \ldots \beta_n$ possibly empty sequences of parentheses, we call β_i ($0 \leq i \leq n$) the i-th *frame* of α. As a consequence, a correctly nexted parenthesis expression consists of only one frame if it has no letter in it. Let $\alpha_0, \ldots, \alpha_n$ and $\beta_0, \ldots, \beta_n$ be the sequences of frames of α and β respectively. If α_k and β_k make the first distinction (that is, k is the greatest index such that $\alpha_i = \beta_i$ for any $0 \leq i < k$), then the index k is called the *fork* of α and β.

For each parse tree t in $\mathtt{PT}(r)$, consider a parenthesis expression $\Phi'(p, t)$ defined

as follows:

$$\begin{aligned}
\Phi'(p, 1) &= {}_{(p}\ )_p \\
\Phi'(p, a) &= {}_{(p}a\ )_p \quad \text{where } a \in \Sigma \\
\Phi'(p, t_1{\cdot}t_2) &= {}_{(p}\Phi'(p.1, t_1)\ \Phi'(p.2, t_2)\ )_p \\
\Phi'(p, \mathtt{L}(t)) &= {}_{(p}\Phi'(p.1, t)\ )_p \\
\Phi'(p, \mathtt{R}(t)) &= {}_{(p}\Phi'(p.2, t)\ )_p \\
\Phi'(p, \mathtt{I}(t_1, \ldots, t_n)) &= {}_{(p}\mathtt{str}(t_1)\ldots\mathtt{str}(t_{n-1})\Phi(p.1, t_n)\ )_p
\end{aligned}$$

where $\mathtt{str}(t)$ is the string derived from t. For a parse configuration $\langle u, t, v\rangle$ where $t \in \mathtt{PT}(r)$, we define $\Phi'(\langle u, t, v\rangle) = u\ \Phi'(\Lambda, t)\ v$.

Lemma 6 *Let r be a regular expression and $c \in \mathtt{PC}(r)$. c captures an interval (i, j) at a position p if and only if ${}_{(p}$ (resp. $)_q$) occurs in the i-th (resp. j-th) frame of $\Phi'(c)$.*

3.2 Position NFAs with Augmented Transitions

A *position automaton with augmented transitions* (PAT, in short) is a 6-tuple

$$\langle \Sigma, Q, T, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$$

where Σ is a finite set of letters, Q a finite set of *states*, T a finite set of *tags*, Δ a subset of $Q \times \Sigma \times Q \times T^*$, $\mathtt{q}_\wedge$ an initial state in Q and $\mathtt{q}_\$$ a final state in Q. We call an element of Δ a *transition*.

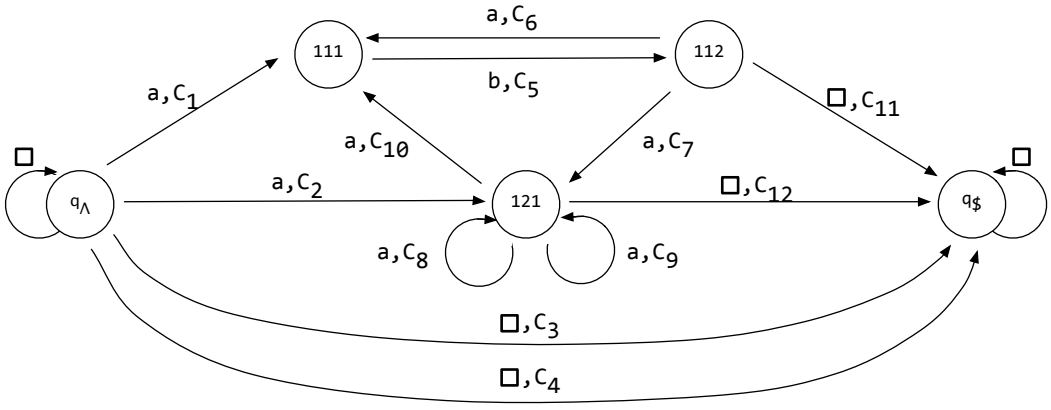
For a PAT $M = \langle \Sigma, Q, T, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$ \rangle$ and an input string w, a *configuration* is a triple $\langle u, p, \alpha\rangle$ where u is a suffix of w, p a state in Q, and α a sequence of tags. For a transition $\delta = \langle p_1, a, p_2, \tau\rangle$ in Δ, we write $\langle u_1, p_1, \alpha_1\rangle \vdash_\delta \langle u_2, p_2, \alpha_2\rangle$ if $u_1 = au_2$ and $\alpha_2 = \alpha_1\tau a$. The *initial* configuration is $\langle w, \mathtt{q}_\wedge, \varepsilon\rangle$ (where ε denotes the empty sequence), while a *finial* configuration is of the form $\langle \varepsilon, \mathtt{q}_\$, \alpha\rangle$ for some α. The subscript of $\vdash$ is sometimes omitted.

Given an input string w such that $w = uv$ for some strings u, v, we say a PAT M *reaches* to a state q in n-steps, *yielding* a sequence α (for a prefix u of w) if there exists a sequence of length (i.e., the number of $\vdash$) n such that $\langle uv, \mathtt{q}_\wedge, \varepsilon\rangle \vdash \cdots \vdash \langle v, q, \alpha\rangle$ where $uv = w$. Moreover, we say a PAT M *accepts* the string u if $v = \varepsilon$ and $q = \mathtt{q}_\$$.

Let r be a regular expression, and p a position. The PAT $M(r, p)$ for r with respect to p is recursively constructed according to the structure of r as follows:

1. $M(1, p)$ is $\langle \Sigma, \{\mathtt{q}_\wedge, \mathtt{q}_\$\}, \{{}_{(p},)_p\}, \{\langle \mathtt{q}_\wedge, a, \mathtt{q}_\$, {}_{(p})_p\rangle \mid a \in \Sigma\}, \mathtt{q}_\wedge, \mathtt{q}_\$\rangle$.

2. For $a \in \Sigma$, $M(a, p)$ is $\langle \Sigma, \{\mathtt{q}_\wedge, p, \mathtt{q}_\$\}, \{{}_{(p},)_p\}, \Delta, \mathtt{q}_\wedge, \mathtt{q}_\$\rangle$ where Δ consists of the transitions $\langle \mathtt{q}_\wedge, a, p, {}_{(p}\rangle$ and $\langle p, b, \mathtt{q}_\$,)_p\rangle$ for any $b \in \Sigma$.



C_1: (∧,(1,(11,(111
C_2: (∧,(1,(12,(121
C_3: (∧,)∧
C_4: (∧,(1,(12,)12,)1,)∧
C_5:)111,(112
C_6:)112,)11,)1,(1,(11,(111
C_7:)112,)11,)1,(1,(12,(121
C_8:)121,(121
C_9:)121,)12,)1,(1,(12,(121
C_{10}:)121,)12,)1,(1,(11,(111
C_{11}:)112,)11,)1,)∧
C_{12}:)121,)12,)1,)∧

Figure 1: The position automata with augmented transitions generated from a regular expression (a·b+a∗)∗

3. If $M(r_i, p.i)$ is $\langle \Sigma, Q_i, T_i, \Delta_i, q_\wedge, q_\$ \rangle$ for $i = 1, 2$, then $M(r_1+r_2, p)$ is

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{ (_p,)_p \}, [\Delta_1 \cup \Delta_2]_p, q_\wedge, q_\$ \rangle$$

4. If $M(r_i, p.i)$ is $\langle \Sigma, Q_i, T_i, \Delta_i, q_\wedge, q_\$ \rangle$ for $i = 1, 2$, then $M(r_1 \cdot r_2, p)$ is

$$\langle \Sigma, Q_1 \cup Q_2, T_1 \cup T_2 \cup \{ (_p,)_p \}, [\Delta_1 \cdot \Delta_2]_p, q_\wedge, q_\$ \rangle.$$

5. If $M(r, p.1)$ is $\langle \Sigma, Q, T, \Delta, q_\wedge, q_\$ \rangle$, then $M(r*, p)$ is $\langle \Sigma, Q, T \cup \{ (_p,)_p \}, [\Delta*]_p, q_\wedge, q_\$ \rangle$.

Finally, for $M(r, \Lambda) = \langle \Sigma, Q, T, \Delta, q_\wedge, q_\$ \rangle$, we define the PAT $M(r)$ for a regular expression r as $\langle \Sigma, Q, T, \Delta \cup \Delta_0, q_\wedge, q_\$ \rangle$ where Δ_0 consists of the transitions $\langle q_\wedge, a, q_\wedge, \varepsilon \rangle$ for any $a \in \Sigma$ and $\langle q_\$, a, q_\$, \varepsilon \rangle$ for any $a \in \Sigma$.

In the above construction, $[\Delta]_p$, $\Delta_1 \cdot \Delta_2$ and $\Delta*$ are respectively given as follows:

$$\begin{aligned}[\Delta]_p = {} & \{ \langle q_1, a, q_2, \tau \rangle \in \Delta \mid q_1 \neq q_\wedge, q_2 \neq q_\$ \} \cup \{ \langle q_\wedge, a, q_\$, (_p \tau)_p \rangle \mid \langle q_\wedge, a, q_\$, \tau \rangle \in \Delta \} \\ & \cup \{ \langle q_\wedge, a, q, (_p \tau \rangle \mid \langle q_\wedge, a, q, \tau \rangle \in \Delta, q \neq q_\$ \} \\ & \cup \{ \langle q, a, q_\$, \tau)_p \rangle \mid \langle q, a, q_\$, \tau \rangle \in \Delta, q \neq q_\wedge \}\end{aligned}$$

$$\Delta_1 \cdot \Delta_2 = \{\langle q_1, a, q_2, \tau\rangle \in \Delta_1 \mid q_2 \neq \mathrm{q}_{\$}\} \cup \{\langle q_1, a, q_2, \tau\rangle \in \Delta_2 \mid q_1 \neq \mathrm{q}_{\wedge}\}$$
$$\cup \{\langle q_1, a, q_2, \tau_1\tau_2\rangle \mid \langle q_1, \_, \mathrm{q}_{\$}, \tau_1\rangle \in \Delta_1, \langle \mathrm{q}_{\wedge}, a, q_2, \tau_2\rangle \in \Delta_2\}$$

$$\Delta* = \Delta \cup \{\langle \mathrm{q}_{\wedge}, a, \mathrm{q}_{\$}, \varepsilon\rangle \mid a \in \Sigma\}$$
$$\cup \{\langle q_1, a, q_2, \tau_1\tau_2\rangle \mid \langle q_1, \_, \mathrm{q}_{\$}, \tau_1\rangle \in \Delta, \langle \mathrm{q}_{\wedge}, a, q_2, \tau_2\rangle \in \Delta, q_1 \neq \mathrm{q}_{\wedge}, q_2 \neq \mathrm{q}_{\$}\}$$

Note that the set T of tags in $M(r)$ is given by the parentheses, each of which has a position as its index, so that $M(r)$ yields parenthesis expressions.

Fig. 1 shows the PAT $M((\mathtt{a}\cdot\mathtt{b}+\mathtt{a}*)*)$ obtained by the above construction, where the symbol $\Box$ stands for arbitrary letters in Σ; that is, a transition with $\Box$ actually represents several transitions obtained by replacing $\Box$ with a letter in Σ.

Notice that the PATs constructed above requires a *look-ahead* symbol. We assume that Σ includes an extra symbol \$ for indicating the "end of string," and that any string given to a PAT $M(r)$ has a trailing \$.

We denote the set of parenthesis expressions that a PAT M yields by accepting an input string $w\$$ as $\mathtt{PE}(M, w)$. To be precise, we ignore the trailing \$ in resulting expressions; i.e., $\mathtt{PE}(M, w) = \{\alpha \mid M \text{ accepts } w\$, \text{ yielding } \alpha\$\}$. Note that $\mathtt{PE}(M(r), w)$ is obtained by appending an arbtrary string, as a prefix or a suffix; i.e., we have

$$\mathtt{PE}(M(r), w) = \{u\alpha v \mid \alpha \in \mathtt{PE}(M(r, \Lambda), w'),\ u, w', v \in \Sigma^* \text{ such that } uw'v = w\}.$$

We also write

$$\mathtt{PE}(M(r, p)) = \bigcup_{w\in\Sigma^*} \mathtt{PE}(M(r, p), w) \qquad \mathtt{PE}(M(r)) = \bigcup_{w\in\Sigma^*} \mathtt{PE}(M(r), w)$$

It is not difficult to see, from the construction of PAT's, that the following equations are satisfied:

Lemma 7

$$\begin{aligned}
\mathtt{PE}(M(1, p)) &= \{\ (_p\,)_p\} \\
\mathtt{PE}(M(a, p)) &= \{\ (_p a\,)_p\} \\
\mathtt{PE}(M(r_1{\cdot}r_2, p)) &= \{\ (_p \alpha_1\alpha_2\,)_p \mid \alpha_i \in \mathtt{PE}(M(r_i, p.i)) \textit{ for } i = 1, 2\} \\
\mathtt{PE}(M(r_1{+}r_2, p)) &= \{\ (_p \alpha_1\,)_p \mid \alpha_1 \in \mathtt{PE}(M(r_1, p.1))\} \cup \{\ (_p \alpha_2\,)_p \mid \alpha_2 \in \mathtt{PE}(M(r_2, p.2))\} \\
\mathtt{PE}(M(r*, p)) &= \bigcup_{n\geq 0} \{\ (_p \alpha_1 \ldots \alpha_n\,)_p \mid \alpha_1, \ldots, \alpha_n \in \mathtt{PE}(M(r, p.1)), \\
&\qquad\qquad \textit{and } \| a_1 \| > 0, \ldots, \| a_n \| > 0 \textit{ if } n \geq 2\ \}
\end{aligned}$$

Lemma 8 $\mathtt{CPT}_{\Phi}(r, p) = \mathtt{PE}(M(r, p))$ *for any regular expression r and position p.*

PROOF The proof is based on the structural induction on the given regular expression r with PROPOSITION 2 and LEMMA 7. □

The following theorem states that a PAT is capable of exactly enumerating any, and only canonical parse configurations:

Theorem 2 $\mathtt{PC}_{\Phi}(r) = \mathtt{PE}(M(r))$ *for any regular expression* r.

PROOF From LEMMA 8, the following equation can be derived:

$$\begin{aligned}\mathtt{PC}_{\Phi}(r) &= \{u\alpha v \mid \alpha \in \mathtt{CPT}_{\Phi}(r),\ u, v \in \Sigma^*\} \\ &= \{u\alpha v \mid \alpha \in \mathtt{PE}(M(r, \Lambda)),\ u, v \in \Sigma^*\} \\ &= \mathtt{PE}(M(r), w).\end{aligned}$$

Corollary 2 $\mathtt{PC}_{\Phi}(r, w) = \mathtt{PE}(M(r), w)$ *for any regular expression* r *and string* w.

4 Developing a Matching Algorithm

4.1 Basic Idea

As mentioned before, our matching algorithm basically emulates the subset construction on the fly. The only but crucial difference is that we need to choose, in each step, the most prior one when paths converge on the same state.

The structure of a parse tree is represented in a parenthesis expression as nesting structure, so that the norm of a subtree corresponds with the distance, the number of frames, from a opening parenthesis to the corresponding closing parenthesis. The problem here is how to reduce the efforts of comparing those distances as well as possible.

For this, we focus on the first step at which two paths are branching. Our key observation is that the opening parentheses that occur after the branching can be entirely ignored. We have already seen in a previous chapter, at least one of the first disagreement pair of parentheses is always an opening parenthesis, and this open parenthsis corresponds with a subtree which trivially has a larger norm since its counterpart has the least norm -1; Since open parentheses occur in order of precedence, we can safely ignore the open parentheses that occur after this branching. This observation considerably reduces the number of parentheses to be considered. Moreover, since the opening parentheses in consideration occur exactly at the same position in the common prefix of those paths, we can compare the norms by just looking at the position of closing parentheses.

Further reduction is possible. Some open parentheses in the common prefix have already been closed before the branching, and hence can be ignored, too.

The open parentheses to be compared are easily identified by using stack. Consider *imaginary* stacks, one for each path. Along each path, opening parentheses are pushed on the stack in order of occurrence, and are eventually removed when the corresponding closing parentheses are encountered. Then, the contents of stacks at the moment of branching indicates the open parentheses to be compared. To distinguish this content, we prepare, for each stack, another stack pointer that we call *bottom* pointer, which initially designates the top of the content (equivalently, the bottom of the forthcoming parentheses) and decreases when the corresponding closing parentheses are encountered. Comparing the bottom pointers at each step allows us to know exactly when the corresponding closing parentheses appear.

Actually, the particular content of each stack does not matter since we already know that parentheses are correctly nested; what's really important is the minimum record of each bottom pointer that have ever achieved within the steps from the beginning to the current step. Moreover, we no longer need stack (top) pointers since they are indicated by the index of a parenthesis.

The above consideration allows us to develop a rather simple way of compar-

ing the priority of paths only based on operations of the bottom pointers, without concerning each of particular parentheses at runtime. We fomarlize this idea in Section 4.2 below.

4.2 Formalization

First, we define the *height* of an opening parenthesis $(_p$ as $|p|+1$ while the height of a closing parenthesis $)_p$ as p. Intuitively, the height of a parenthesis is the value of the (imaginary) stack pointer immediately after processing the parenthesis, which we discussed above.

For any sequences α and β of parentheses, $\alpha \sqcap \beta$ denotes the longest common prefix of α and β. For any sequence α of parentheses and a prefix α' of α, $\alpha \backslash \alpha'$ denotes the remaining sequence obtained by removing α' from α. In case $\alpha \backslash (\alpha \sqcap \beta)$ is non-empty, we denote the first element of $\alpha \backslash (\alpha \sqcap \beta)$ as α / β. In other words, α / β is the first (that is, the leftmost) symbol in α which does not appear in β. Obviously, at least one of α / β or β / α exists if $\alpha \neq \beta$.

Let α and β be parenthesis expressions whose frames are $\alpha_0, \ldots, \alpha_n$ and $\beta_0, \ldots, \beta_n$ respectively. Suppose that k is the fork (i.e., the index of the first different frames) of α and β.

We define the *trace* of α with respect to β, written $\mathtt{tr}_\beta(\alpha)$, as a sequence $\rho_0, \ldots, \rho_n$ of integers as follows:

$$\rho_i = \begin{cases} -1 & (i < k) \\ \min\{\mathtt{lastsp}(\alpha_k \sqcap \beta_k), \mathtt{minsp}(\alpha_k \backslash (\alpha_k \sqcap \beta_k))\} & (i = k) \\ \min\{\rho_{i-1}, \mathtt{minsp}(\alpha_i)\} & (i > k) \end{cases}$$

where $\mathtt{lastsp}(\gamma)$ denotes the height of the last (that is, the rightmost) parenthesis in γ, while $\mathtt{minsp}(\gamma)$ is the minimal height of the parentheses in γ for non-empty γ; we define $\mathtt{lastsp}(\gamma) = \mathtt{minsp}(\gamma) = 0$ if γ is empty. Intuitively, The i-th value ρ_i of $\mathtt{tr}_\beta(\alpha)$ ($i > k$ where k is the fork) means the minimal record of the bottom pointer for α within the steps from the fork to the i-th step. The negative number -1 just means that the bottom pointer is not yet set. We denote the initial value $\min\{\mathtt{lastsp}(\alpha \sqcap \beta), \mathtt{minsp}(\alpha \backslash (\alpha \sqcap \beta))\}$ as $\mathtt{bp0}_\beta(\alpha)$.

Usually, it is convenient to treat $\mathtt{tr}_\beta(\alpha)$ and $\mathtt{tr}_\alpha(\beta)$ together. We write

$$\mathtt{tr}\begin{bmatrix}\alpha \\ \beta\end{bmatrix} = \begin{bmatrix}a_0 \\ b_0\end{bmatrix} \cdots \begin{bmatrix}a_n \\ b_n\end{bmatrix},$$

or more concisely, $\mathtt{tr}(\alpha;\beta) = (a_0, a_0), \ldots, (b_n, b_n)$ where $a_0, \ldots, a_n$ and $b_0, \ldots, b_n$ are respectively $\mathtt{tr}_\beta(\alpha)$ and $\mathtt{tr}_\alpha(\beta)$. Likewise, $\mathtt{bp0}(\alpha;\beta)$ denotes $(\mathtt{bp0}_\beta(\alpha), \mathtt{bp0}_\alpha(\beta))$.

Let α and β be parenthesis expressions. For $\mathtt{tr}(\alpha;\beta) = (\rho_0, \rho'_0), \ldots, (\rho_n, \rho'_n)$, we write $\alpha \sqsubset \beta$ if $\rho_i > \rho'_i$ for the least i such that $\rho_j = \rho'_j$ for any $j > i$. We write $\alpha \sim \beta$

if $\rho_i = \rho'_i$ for any $0 \leq i \leq n$. In particular, we immediatelly obtain $\alpha \sim \beta$ if α and β do not have any non-empty common prefix.

Let α and β be correctlly nested parenthesis expressions, k their fork, and α' (resp. β') is the k-th frame of α (resp. β). We write $\alpha \sqsubset'_k \beta$, if the following conditions, (1) and (2), hold for some position p; (1) there exists $\alpha'/\beta' = (_p$; (2) if there exists $\beta'/\alpha' = (_q$ for some q then $p \lessdot q$. We simply write $\alpha \sqsubset' \beta$ if $\alpha \sqsubset'_k \beta$ for some k.

Definition 9 *Let α and β be parenthesis expressions. We say α is* prior to *β, written $\alpha \lessdot \beta$, if either $\alpha \sqsubset \beta$ or else $\alpha \sim \beta$ and $\alpha \sqsubset' \beta$.*

Note that parenthesis expressions can be compared only if they have the same number of frames (or equivalently, the same number of letters).

In the following, we will prove that the relation $\lessdot$ we have just defined is indeed an ordering that is essentially the same as the precedence $\lessdot$ on parse trees that we have previously introduced.

Our next aim is to show the compatibility of the above definition with the basic definition of priority in Section 2. We begin with a few auxiliary propositions, all of which immediately follow from the above definition.

Lemma 9 *The relation $\lessdot$ on $\mathtt{PC}_\Phi(r, w)$ is anti-symmetric; i.e., $s \lessdot t$ implies $t \not\lessdot s$ for any s and t in $\mathtt{PC}_\Phi(r, w)$.*

PROOF Obvious from DEFINITION 9. □

Lemma 10 *Let r be a regular expression, w a string, and $s, t \in \mathtt{CPT}(r, w)$ such that $s \lessdot t$. Let q be the first distinct position of s and t. If $|q| = 1$, then both $\Phi^p_r(s) \sim \Phi^p_r(t)$ and $\Phi^p_r(s) \sqsubset' \Phi^p_r(t)$ holds for any p.*

PROOF Since $|q| = 1$, q is an integer, say, i. We have two cases:

[1] In case $r = r_1 + r_2$, s and t are respectively of the forms $\mathtt{L}(s_1)$ and $\mathtt{R}(t_1)$ for some s_1 and t_1. Then, we have $\Phi^p_r(s) = (_p \Phi^{p.1}_{r_1}(s_1))_p$ and $\Phi^p_r(t) = (_p \Phi^{p.2}_{r_2}(t_1))_p$. Since $\| s_1 \| = \| t_1 \|$, we have $\Phi^p_r(s) \sim \Phi^p_r(t)$.

[2] In case $r = r_1*$, PROPOSOTION 1 tells us that s and t are respectively of the forms $\mathtt{I}(u_1, \ldots, u_{i-1}, s_i)$ and $\mathtt{I}(u_1, \ldots, u_{i-1})$ where the sequence $u_1, \ldots, u_{i-1}$ is possibly empty. Since $\| s_i \| = 0$, it is not difficult to see that $\Phi^p_r(s) \sim \Phi^p_r(t)$.

In both cases, $\Phi^p_r(s) \sqsubset' \Phi^p_r(t)$ is obviously true. □

Lemma 11 *Let r be a regular expression, w a string, and $s, t \in \mathtt{CPT}(r, w)$ such that $s \lessdot_q t$. If $|q| = 1$ and q is not the first distinct position, then $\Phi^p_r(s) \sqsubset \Phi^p_r(t)$ holds for any p.*

PROOF Since $|q| = 1$, q is an integer, say, i. PROPOSITION 1 lets us know that s and t are respectively of the forms: $f(u_1, \ldots, u_{i-1}, s_i, \ldots, s_n)$ and $f(u_1, \ldots, u_{i-1}, t_i, \ldots, t_{n'})$ where f is either $\cdot$ or $\mathtt{I}$, and the common sequence $u_1, \ldots, u_{i-1}$ is possibly empty. Since $\| s_i \| > \| t_i \|$ but $\| s \| = \| t \|$, it is not difficult to see that $\Phi_r^p(s) \sqsubset \Phi_r^p(t)$. □

Lemma 12 *Let r be a regular expression, w a string, and $s, t \in \mathtt{CPT}(r, w)$ such that $\Phi_r^{p.i}(s|_i) \lessdot \Phi_r^{p.i}(t|_i)$ for some integer position i. If $s|_k = t|_k$ for any integer $k < i$ such that $k \in \mathtt{Pos}(s) \cup \mathtt{Pos}(t)$, then $\Phi_r^p(s) \lessdot \Phi_r^p(t)$.*

PROOF Let $\alpha = \Phi_r^p(s)$ and $\beta = \Phi_r^p(t)$. They are given by

$$\alpha = {}_p\gamma_1 \ldots \gamma_{i-1}\alpha_i \ldots \alpha_n{}_p,$$
$$\beta = {}_p\gamma_1 \ldots \gamma_{i-1}\beta_i \ldots \beta_{n'}{}_p$$

where $\gamma_k = \Phi_r^{p.k}(s|_k) (= \Phi_r^{p.k}(t|_k))$ $(k < i)$, $\alpha_k = \Phi_r^{p.k}(s|_k)$ $(i \leq k \leq n)$, $\beta_k = \Phi_r^{p.k}(t|_k)$ $(i \leq k \leq n')$, and the common sequence $\gamma_1 \ldots \gamma_{i-1}$ may be empty. Hence, $\alpha_i \sqsubset' \beta_i$ implies $\alpha \sqsubset' \beta$. It is also not difficult to see $\alpha \sqsubset \beta$ when $\alpha_i \sqsubset \beta_i$. □

The following key lemma states that the function Φ is order-preserving:

Lemma 13 *Let r be a regular expression, w a string, and $s, t \in \mathtt{CPT}(r, w)$. For any p, $s \lessdot t$ implies $\Phi_r(s, p) \lessdot \Phi_r(t, p)$.*

PROOF The proof is based on structual induction on r. Let q be the first distinct position of s and t. We distinguish two cases according with the length of $q(> 0)$. [1] In case $|q| = 1$, the result follows by LEMMA 10. [2] Otherwise, let q' be the decision position of s and t. If $|q'| = 1$, the result follows by LEMMA 11. Otherwise, put q' as $i.q''$ for some integer i. Since $s|_i \lessdot t|_i$, We have, by induction hypothesis, $\Phi_{r|_j}^{p.i}(s|_i) \lessdot \Phi_{r|_j}^{p.i}(t|_i)$ for any p where $j = \mathtt{fold}(i)$. Since PROPOSITION 1 tells us $s|_k = t|_k$ for any $k < i$ such that $k \in \mathtt{Pos}(s) \cup \mathtt{Pos}(t)$, the result follows by LEMMA 12. □

The converse is also true:

Lemma 14 *Let r be a regular expression, w a string, and $s, t \in \mathtt{CPT}(r, w)$ such that $\| s \| = \| t \|$. For any p, $\Phi(s, p) \lessdot \Phi(t, p)$ implies $s \lessdot t$.*

PROOF Suppose $\Phi(s) \lessdot \Phi(t)$ but $s \not\lessdot t$. Since $\Phi(s)$ and $\Phi(t)$ are different, s and t is not identical. Hence, by THEOREM 1, we have $t \lessdot s$, which implies $\Phi(t) \lessdot \Phi(s)$.

However, this contradicts the fact that $\lessdot$ is asymmetric on the set of parenthesis expressions (LEMMA **??**).

We conclude that the stepwise comparison is compatible with the basic definition of priority:

Lemma 15 *Let r be a regular expression, and w a string. For any $u, v \in \mathsf{PC}(r, w)$ we have $u \lessdot v$ if and only if $\Phi(u) \lessdot \Phi(v)$.*

PROOF ($\Rightarrow$) Consider $u = \langle u_1, s, u_2 \rangle$ and $v = \langle v_1, t, v_2 \rangle$ in $\mathsf{PC}(r, w)$ such that $u \lessdot v$. We distinguish two cases.

[1] Suppose $|u_1| < |v_1|$. We have $\Phi(u) = u_1 \Phi(s) u_2$ and $\Phi(v) = v_1 \Phi(s) v_2$.

($\Leftarrow$)

The order we have just defined above is essentially the same as the priority order on parse configurations:

Theorem 3 *Let r be a regular expression, and w a string. $\langle \mathsf{PC}_\Phi(r, w), \lessdot \rangle$ is an order set isomorphic to $\langle \mathsf{PC}(r, w), \lessdot \rangle$.*

PROOF Strict totality has been shown in PROPOSOTION **??**. Transitivity of $\langle \mathsf{PC}_\Phi(r, w), \lessdot \rangle$ follows from LEMMA 15. Irreflexivity is an consequence of transitivity and asymmetry (LEMMA **??**).

Next, we show that Φ gives a bijection from $\mathsf{PC}(r, w)$ to $\mathsf{PC}_\Phi(r, w)$. This means that $\mathsf{PC}(r, w)$ and $\mathsf{PC}_\Phi(r, w)$ are isomorphic since they are finite total order sets. Consider $s, t \in \mathsf{CPT}(r, w)$ such that $s \neq t$. By THEOREM **??**, we have either $s \lessdot t$ or $t \lessdot s$. Without loss of generarity, we assume $s \lessdot t$, which implies $\Phi(s) \lessdot \Phi(t)$ by LEMMA 15. Finally, we have $\Phi(s) \neq \Phi(t)$ by irreflexivity.

Algorithm 1 Match a string w against a regular expression r

$K := \{q_\wedge\}$
Let $P[q_\wedge]$ point an empty list
for all $n := 0, \ldots, |w|$ **do**
  Read the next letter a in $w\$$ (left to right)
  `proceed_one_step`$(n,$ `effective_transitions`$(a))$
  if K contains the final state $q_{\$}$ **then**
    From K drop $q_\wedge$ and all q such that $L[q] > L[q_{\$}]$
    break if K contains only $q_{\$}$
  end if
end for
if K contains the final state $q_{\$}$ **then**
  Report the captured positions via `report`$(P[q_{\$}])$
else
  Report FAILURE
end if

5 Matching Algorithm and Its Correctness

Algorithm 2 `effective_transitions`(a)

$trans := \emptyset$
for all state $q \in Q_a$ such that `tag`$(p, q) \neq \perp$ for some $p \in K$ **do**
  $K' := K \backslash \{p\}$
  for all state $p' \in K'$ such that `tag`$(p', q) \neq \perp$ **do**
    $\rho :=$min$\{B[p][p'],$`minsp`$($`tag`$(p, q))\}$; $\rho' :=$min$\{B[p'][p],$`minsp`$($`tag`$(p', q))\}$;
    $p := p'$ **if** $\rho < \rho'$ or $\rho = \rho'$ and $D[p'][p] = 1$
  end for
  Add $\langle p, q,$ `tag`$(p, q)\rangle$ to $trans$
end for
return trans

Based on the above discussion, we provide a regular expression matching algorithm. Algorithm 1 shows the main routine of the algorithm, which takes, apart from a PAT $M(r)$ built from a regular expression r, an input string $w\$$ then answers whether w can match against r or not. If the matching succeeds, the algorithm tells us the positions of substrings captured by subexpressions.

Throughout the entire algorithm, a couple of global variables are maintained: K, sp, B, D, P and L. K stores the set of current states. Let α and β be the paths getting to states p and q in K respectively. Then, $B[p][q]$ designates the stack bottom pointers for α and β. $D[p][q]$ remembers which path is prior to the other. $P[p]$ stores a pointer to the list, each element of which is a sequence of tags in T_{cap}

Algorithm 3 `proceed_one_step`$(n, trans)$

$K := \{q \mid \langle \_, q, \_ \rangle \in trans\}$
Create copies D', P', B' of D, P, B respectively.
for all $\langle p, q, \alpha \rangle, \langle p', q', \alpha' \rangle \in trans$ such that $q \lessdot q'$ **do**
  if $p = p'$ **then**
    $D[q][q'] := 1$ **if** $\alpha \sqsubset' \alpha'$; $D[q][q'] := -1$ **if** $\alpha' \sqsubset' \alpha$
    $\rho := \mathtt{bp0}_{\alpha'}(\alpha)$; $\rho' := \mathtt{bp0}_{\alpha}(\alpha')$
  else
    $D[q][q'] := D'[p][p']$
    $\rho := \min\{B'[p][p'], \mathtt{minsp}(\alpha)\}$; $\rho' := \min\{B'[p'][p], \mathtt{minsp}(\alpha')\}$;
  end if
  $D[q][q'] := 1$ **if** $\rho > \rho'$; $D[q][q'] := -1$ **if** $\rho < \rho'$; $D[q'][q] := -D[q][q']$
  $B[q][q'] := \rho$; $B[q'][q] := \rho'$;
end for
for all $\langle p, q, \alpha \rangle \in trans$ **do**
  $P[q] := P'[p]$
  Add α' to the end of $P[q]$ where α' contains tags t in α such that $t \in T_{cap} \cup T_*$
  $L[q] := n$ **if** α contains $($$_\Lambda$
end for

Algorithm 4 `report`(ℓ)

for each tag t in the m-th element of ℓ (in reverse order) **do**
  if t is a closed parenthesis ever seen before, say $)_p$ **then**
    Skip until the corresponding tag $($$_p$ occurrs, and continue to the next tag
  else
    Report m as the captured position for t if $t \in T_{cap}$.
    Remember t if it is a closed tag
  end if
end for

or T_* appearing in each frame in α. Finally, $L[q]$ stores the position at which the parenthesis $(_\Lambda$ appears in α.

The main routine calls two other subroutines; `effective_transitions`(a) (Alg. 2) takes a letter a of the input string then returns a set of transitions actually used for that step, pruning less prior, thus ineffective, transitions, while `proceed_one_step`$(n, trans)$ (Alg. 3) updates the values of the global variables for the next step of computation.

In case Algorithm 1 succeeds, we call $\{P[q_\$].buf[0], \ldots, P[q_\$].buf[N-1]\}$ a *match* where N be the latest value of $P[q_\$].idx$.

Theorem 4 *Let r be a regular expression,w a string and C be a set of capturing positions in r. If* `PC`(r, w) *is non-empty, the algorithm 1 succeeds. Otherwise, the algorithm 1 fails. In the former case, let c be the least parse configuration in* `PC`(r, w)*, and M the match. Then, M include $(\ (_p, i)$ and $(\ )_p, j)$ if and only if c captures an interval (i, j) at the position p.*

Since `bp0`$()$, `minsp`$()$, and the ordering $\sqsubset$ can be computed, in advance, at compile time, namely, before a particular input string is given, the time complexity of `effective_transitions`$()$ is $O(n^2)$, where n is the number of occurrences of the most frequently used letter in a given regular expression and $O(n(n+c))$ for `proceed_one_step`$()$ where c is the number of capturing positions plus the number of $*$-nodes. Therefore, for the length m of a input string, the time complexity of the whole algorithm is given as follows:

Theorem 5 *The time complexity of the algorithm is $O(mn(n+c))$.*

6 Concluding Remarks

We have presented a regular expression matching algorithm that follows the leftmost-longest rule. We also have been pursuing the correctness of the algorithm by offering a formalization of the rule based on canonical parse trees. The worst case computational cost, $O(mn(n+c))$, is acceptable in most practical applications because n and c remains rather small, while the length m of an input string could increase substantially in most cases. One controversial aspect of our formal interpretation of the leftmost-longest rule is whether it is acceptable for a common understanding of the community. For this, we state that it is consistent with an extensively accepted interpretation [5] which investigates the POSIX specification very closely.

The idea of realizing regular expression matching by emulating subset construction at runtime goes back to early days; see [2] for the history. The idea of using tags for representing the syntactical structure of regular expression is indebted to the study [10], in which Laurikari have introduced *tagged* automata (the basis of TRE library) in order to formulate the priority of paths. Unfortunately, it is somewhat incompatible with today's interpretation [5] as for the treatment of repetition.

Dubé and Feeley have given a way of generating the entire set of parse trees from regular expressions [4] by using a grammatical representation. Frisch and Cardelli have also considered an ordering on parse trees [6], which is completely different from ours since they focus on the greedy, or first match, semantics. They also focus on a problem of ε-transition loop, which does not occur in our case since we are based on position automata. Vansummeren [13] have also given a stable theoretical framework for the leftmost-longest matching, although capturing inside repetitions is not considered.

An implementation taking a similar approach to ours is Kuklewicz's Haskell TDFA library [9]. Although it is also based on position automata, the idea of using *orbit tags* for the comparison of paths is completely different from our approach. Another similar one is Google's new regular expression library called RE2 [3] which has come out just before we finish the preparation of this paper. TRE, TDFA, RE2 and our algorithm are all based on automata, so that, while their scope is limited to regular expressions without *back references*, they all enable of avoiding the computational explosion.

Acknowledgements: This work is supported by Japan Society for Promotion of Science, Basic Research (C) No.22500019.

References

[1] F. Baader and T.Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] R. Cox. Regular expression matching can be simple and fast. Available online: `http://swtch.com/~rsc/regexp/regexp1.html`, 2007.

[3] R. Cox. Regular expression matching in the wild. Available online: `http://swtch.com/~rsc/regexp/regexp3.html`, 2010.

[4] D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta Infomatica*, 37(2):121–144, 2000.

[5] G. Fowler. An interpretation of the POSIX regex standard. Available online: `http://www2.research.att.com/~gsf/testregex/re-interpretation.html`, 2003.

[6] A. Frisch and L. Cardelli. Greedy regular expression matching. In *ICALP04 (LNCS 3142)*, pages 618–629, 2004.

[7] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[8] The IEEE and The Open Group. The open group base specification Issue 6 IEEE Std 1003.1 2004 Edition. Avaiable online:`http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html`, 2004.

[9] C. Kuklewicz. Regular expressions / Bounded space proposal. Available online:`http://www.haskell.org/haskellwiki/Regular_expressions/Bounded_space_proposal`, 2007.

[10] V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Helsinki University of Technology, 2000.

[11] R. McNaughton and H Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.

[12] H. Spencer. A regular-expression matcher. In *Software Solutions in C*. Academic Press, 1994.

[13] S. Vansummeren. Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.

A Proof of Theorem 4

Lemma 16 *Let Δ be a set of transitions in $M(r,p)$ for some regular expression r and position p. Then Δ satisfies the property that if distinct transitions $\langle s,a,q_1,\tau_1\rangle$ and $\langle s,a,q_2,\tau_2\rangle$ with $q_1 \neq q_2$ belong to Δ, then both τ_1/τ_2 and τ_2/τ_1 exist and either $\tau_1 \sqsubset' \tau_2$ or $\tau_2 \sqsubset' \tau_1$ holds.*

PROOF Structural induction on r. The base case, where $r = 1$ or $r \in \Sigma$, is obvious. In the induction step, we denote the set of transitions of $M(r_i, p.i)$ by Δ_i $(i = 1,2)$. Note that if Δ satisfies the property then so does $[\Delta]_p$ for any p.

[1] Suppose $r = r_1 \cdot r_2$. It suffices to show that $\Delta_1 \cdot \Delta_2$ satisfies the property. We only consider the following two cases because the others immediately follow from the induction hypothesis: (1) Consider $\langle s,a,q_1,\tau\tau_1\rangle$ and $\langle s,a,q_2,\tau\tau_2\rangle$ obtained from $\langle s,\_,\mathtt{q}_\$,\tau\rangle \in \Delta_1$ and $\langle \mathtt{q}_\wedge,a,q_i,\tau_i\rangle \in \Delta_2$ $(i = 1,2)$. Then $\tau\tau_1 \sqcap \tau\tau_2 = \tau(\tau_1 \sqcap \tau_2)$ holds. Since induction hypothesis gurantees that both τ_1/τ_2 and τ_2/τ_1 exist, so do $\tau\tau_1/\tau\tau_2$ and $\tau\tau_2/\tau\tau_1$ and they are equal to τ_1/τ_2 and τ_2/τ_1 respectively. Thus, from the induction hypothesis, either $\tau\tau_1 \sqsubset' \tau\tau_2$ or $\tau\tau_2 \sqsubset' \tau\tau_1$ holds; (2) Consider $\langle s,a,q_1,\tau_1\rangle \in \Delta_1$ and $\langle s,a,q_2,\tau_2\tau\rangle$ obtained from $\langle s,a,\mathtt{q}_\$,\tau_2\rangle \in \Delta_1$ and $\langle \mathtt{q}_\wedge,a,q_2,\tau\rangle \in \Delta_2$. The induction hypothesis assures both τ_1/τ_2 and τ_2/τ_1 exist. Thus $\tau_1/\tau_2\tau$ and $\tau_2\tau/\tau_1$ are equal to τ_1/τ_2 and τ_2/τ_1 respectively. Therefore, from induction hypothesis, either $\tau\tau_1 \sqsubset' \tau\tau_2$ or $\tau\tau_2 \sqsubset' \tau\tau_1$ holds.

[2] Suppose $r = r_1 + r_2$. We only show $\Delta_1 \cup \Delta_2$ satisfies the property. We only consider the case where $\langle \mathtt{q}_\wedge,a,q_i,\tau_i\rangle$ is from Δ_i $(i = 1,2)$. An easy induction reveals that for any regular expression r and position p the transitions leaving from $\mathtt{q}_\wedge$ in $M(r,p)$ start with $(_p$. Hence τ_i starts with $(_{p.i}$ $(i = 1,2)$. It follows that the property holds for $\Delta_1 \cup \Delta_2$. The other cases immediately follow from the induction hypothesis.

[3] Suppose $r = r_1*$. We show that $[\Delta_1*]_p$ satisfies the property. We only consider the following two cases because the others are proved similary to the case of $r = r_1 \cdot r_2$: (1) Consider $\langle q,a,\mathtt{q}_\$,\tau)_p\rangle$ and $\langle q,a,q',\tau\tau'\rangle$ obtained from $\langle q,a,\mathtt{q}_\$,\tau\rangle$ and $\langle \mathtt{q}_\wedge,a,q',\tau'\rangle$ in Δ_1. We have $\tau)_p/\tau\tau' =)_p$. As shown above, τ' starts with $(_{p.1}$. Thus $\tau\tau'/\tau)_p = (_{p.1}$. Therefore $\tau\tau' \sqsubset' \tau)_p$; (2) Consider $\langle \mathtt{q}_\wedge,a,\mathtt{q}_\$,(_p)_p\rangle$ and $\langle \mathtt{q}_\wedge,a,q,(_p\tau\rangle$ obtained from $\langle \mathtt{q}_\wedge,a,q,\tau\rangle \in \Delta_1$, where $q \neq \mathtt{q}_\$$. Since τ starts with $(_{p.1}$, we have $(_p)_p/(_p\tau =)_p$ and $(_p\tau/(_p)_p = (_{p.1}$. Therefore $(_p\tau \sqsubset' (_p)_p$. □

Lemma 17 *Let r be a regular expression. For any distinct transitions $\langle s,a,q,\tau\rangle$ and $\langle s,a,q',\tau'\rangle$ with $q \neq q'$ in $M(r)$, either $\tau \sqsubset' \tau'$ or $\tau' \sqsubset' \tau$ holds.*

PROOF If $\langle \mathtt{q}_\wedge,a,q,\tau\rangle$ and $\langle \mathtt{q}_\wedge,a,\mathtt{q}_\wedge,\varepsilon\rangle$ with $q \neq \mathtt{q}_\wedge$, by definition we have $\tau \sqsubset' \varepsilon$. Then the result follows from LEMMA 16. □

We show the correctness of the algorithm described by Algorithms 1–3.

Lemma 18 *Let α and β be parenthesis expressions and $\langle \rho, \rho' \rangle$ the last element of $\mathtt{tr}(\alpha; \beta)$. If α and β are written as $\alpha' a \gamma$ and $\beta' a \delta$ respectively with some parenthesis expressions α' and β', a letter a and frames γ and δ then $\alpha \lessdot \beta$ if and only if the one of the following property holds.*

1. $\rho > \rho'$
2. $\rho = \rho'$ *and* $\alpha' \lessdot \beta'$
3. $\rho = \rho'$, $\alpha' = \beta'$ *and* $\gamma \sqsubset' \delta$.

PROOF Let $\alpha' = \alpha_0 a_1 \alpha_1 \dots a_n \alpha_n$ and $\beta' = \beta_0 a_1 \beta_1 \dots a_n \beta_n$ for some $n \geq 0$, and $\mathtt{tr}(\alpha'; \beta') = \langle \rho_1, \rho'_1 \rangle \cdots \langle \rho_n, \rho'_n \rangle$. By definition $\alpha \lessdot \beta$ if and only if one of the following holds.

1. $\rho > \rho'$
2. $\rho = \rho'$ and there exists j with $\rho_j > \rho'_j$ and $\rho_i = \rho'_i$ for any $j < i \leq n$
3. $\rho = \rho'$, $\alpha' = \beta'$ and $\gamma \sqsubset' \delta$
4. $\rho = \rho'$ and there exists a fork $k \leq n$ with $\alpha_k \sqsubset' \beta_k$

The result follows because the disjunction of the second and the fourth conditions is equivalent to $\rho = \rho'$ and $\alpha' \lessdot \beta'$. □

Lemma 19 *Let α and β be parenthesis expressions, γ a sequence of parentheses and $a \in \Sigma$. If $\alpha \lessdot \beta$ then $\alpha a \gamma \lessdot \beta a \gamma$.*

PROOF Since $\alpha \lessdot \beta$, they have the same number of frames more than zero. Let $\langle \rho, \rho' \rangle$ and $\langle \sigma, \sigma' \rangle$ be the last elements of $\mathtt{tr}(\alpha; \beta)$ and $\mathtt{tr}(\alpha a \gamma; \beta a \gamma)$, respectively. Since $\alpha \lessdot \beta$, we have $\sigma = \min(\rho, \mathtt{minsp}(\gamma))$ and $\sigma' = \min(\rho', \mathtt{minsp}(\gamma))$. From LEMMA 18 $\rho \geq \rho'$ follows. We distinguish the following three cases.

- If $\mathtt{minsp}(\gamma) > \rho \geq \rho'$ then $\sigma = \rho \geq \rho' = \sigma'$.
- If $\rho \geq \mathtt{minsp}(\gamma) \geq \rho'$ then $\sigma = \mathtt{minsp}(\gamma) \geq \rho' = \sigma'$.
- If $\rho \geq \rho' \geq \mathtt{minsp}(\gamma)$ then $\sigma = \mathtt{minsp}(\gamma) = \sigma'$.

Hence $\sigma \geq \sigma'$. By LEMMA 18, $\alpha \lessdot \beta$ induces $\alpha a \gamma \lessdot \beta a \gamma$. □

Lemma 20 *Let α, β and γ be sequences of parentheses. If $\alpha \sqsubset' \beta$ and $\beta \sqsubset' \gamma$ then $\alpha \sqsubset' \gamma$.*

PROOF If $\alpha \sqcap \beta$ contains more parentheses than $\beta \sqcap \gamma$, then $(\beta \sqcap \gamma)(\beta/\gamma)$ is a prefix of $\alpha \sqcap \beta$. Hence we have $\alpha \sqsubset' \gamma$. The same holds for the opposite case.

Suppose $\alpha \sqcap \beta = \beta \sqcap \gamma$. Because $\alpha \sqsubset' \beta$ and $\beta \sqsubset' \gamma$, we have $\alpha/\beta = (_{p_1}$ and $\beta/\alpha = \beta/\gamma = (_{p_2}$ with $p_1 \lessdot p_2$. If $\gamma/\beta = (_{p_3}$ then $p_2 \lessdot p_3$. Thus $\alpha \sqsubset' \gamma$ follows from $p_1 \lessdot p_3$; If not, $\alpha \sqsubset' \gamma$ obviously holds. □

Hereafter, r and string w denote a regular expression and a string fixed arbitrary. Let $M(r) = \langle \Sigma, Q, T, \Delta, \mathrm{q}_\wedge, \mathrm{q}_\$ \rangle$. For the sake of convenience, we introduce the following: a_0 stands for ε, a_i denotes the i-th letter in w ($1 \leq i \leq |w|$) and $a_{|w|+1}$ means \$. Thus, $w\$$ is denoted by $a_0 \dots a_{|w|+1}$.

The relation $\vdash^n$ on the configurations is defined as follows: $c \vdash^0 c'$ if $c = c'$; $c \vdash^{n+1} c'$ if there exists a configuration c'' such that $c \vdash^n c''$ and $c'' \vdash c'$. We denote a set of reachable states with n steps by $\mathtt{reach}(n)$, i.e., $\mathtt{reach}(n) = \{q \mid \langle w\$, \mathrm{q}_\wedge, \varepsilon \rangle \vdash^n \langle \_, q, \_ \rangle\}$. The set $\{p \in \mathtt{reach}(n) \mid \exists \tau. \langle p, a_n, q, \tau \rangle \in \Delta\}$ for $n \geq 0$ and $q \in Q$ is denoted by $\mathtt{reach}(n)|_q$. The set of parenthesis expressions obtained by the paths reaching q with n steps is presented by $\mathtt{pe}(q, n)$. Precisely, $\mathtt{pe}(q, n) = \{\alpha \mid \exists v. \langle w\$, \mathrm{q}_\wedge, \varepsilon \rangle \vdash^n \langle v, q, \alpha a_n \rangle\}$. Note that for $q \in Q$ and $n \geq 0$ every element of $\mathtt{pe}(q, n)$, if any, is of the form $a_0\alpha_0 \dots a_{n-1}\alpha_{n-1}$ because we let $a_0 = \varepsilon$. Especially, $\mathtt{pe}(\mathrm{q}_\wedge, 0) = \{\varepsilon\}$ and $\mathtt{pe}(q, 0) = \varnothing$ if $q \neq \mathrm{q}_\wedge$.

Lemma 21 *Let α and β be in $\mathtt{pe}(q, n)$ for some $q \in \mathtt{reach}(n)$ and $n \geq 0$. If $\alpha \neq \beta$ then either $\alpha \lessdot \beta$ or $\beta \lessdot \alpha$ holds.*

PROOF If $\alpha \not\sim \beta$ then by definition $\alpha \lessdot \beta$ or $\beta \lessdot \alpha$ holds. Suppose $\alpha \sim \beta$ and $\alpha \neq \beta$. Since every state in Q can reach $\mathrm{q}_\$$, there exists a parenthesis expression γ such that $\alpha a_n \gamma$, $\beta a_n \gamma \in \mathtt{PE}(M(r), v)$ for some $v \in \Sigma^*$. Therefore, LEMMA **??** yields $\alpha \lessdot \beta$ or $\beta \lessdot \alpha$. □

Lemma 22 *For any state $q \in Q$ and $n \geq 0$, the relation $\lessdot$ between parenthesis expressions is a strict total order on $\mathtt{pe}(q, n)$.*

PROOF Irreflexivity immediately follows from the definition of $\lessdot$. Suppose $\alpha \lessdot \beta$ and $\beta \lessdot \gamma$ for $\alpha, \beta, \gamma \in \mathtt{pe}(q, n)$. Since every state in Q can reach $\mathrm{q}_\$$, there exists a parenthesis expression δ such that $\alpha a_n \delta, \beta a_n \delta, \gamma a_n \delta \in \mathtt{PE}(M(r), w)$. Repetitive applications of LEMMA 19 results in $\alpha a_n \delta \lessdot \beta a_n \delta$ and $\beta a_n \delta \lessdot \gamma a_n \delta$. From THEOREMS 1, 2 and 3, $\langle \mathtt{PE}(M(r), w), \lessdot \rangle$ is a strict total order set. It follows $\alpha a_n \delta \lessdot \gamma a_n \delta$. Suppose that $\alpha \lessdot \gamma$ does not hold. It should not be $\alpha = \gamma$; otherwise, the contradiction occurs because $\alpha \lessdot \beta$ and $\beta \lessdot \gamma = \alpha$. Thus, by LEMMA 21 $\gamma \lessdot \alpha$ holds. Applying LEMMA 19 repeatedly, we have $\gamma\delta \lessdot \alpha\delta$. It contradicts with $\alpha\delta \lessdot \gamma\delta$. Hence $\alpha \lessdot \gamma$. Finally, from LEMMA 21 $\lessdot$ is strict total. □

Lemma 23 *Let $n > 0$. For $\alpha_1 \in \mathtt{pe}(q_1, n)$ and $\alpha_2 \in \mathtt{pe}(q_2, n)$ with $q_1 \neq q_2$, we have either $\alpha_1 \lessdot \alpha_2$ or $\alpha_2 \lessdot \alpha_1$.*

PROOF If $\alpha_1 \sim \alpha_2$ does not hold then the result obviously holds. Suppose $\alpha_1 \sim \alpha_2$. Because $q_1 \neq q_2$ both α_1 and α_2 has a common prefix β yielded by a path from $q_{\wedge}$ to a state p, and there exist transitions $\langle p, a, p_i, \tau_i \rangle$ $(i = 1, 2)$ where $\beta\tau_i$ is a prefix of α_i. Then, from LEMMA 17, either $\tau_1 \sqsubset' \tau_2$ or $\tau_2 \sqsubset' \tau_1$ holds. Therefore, we immediately obtaine the desired result. □

LEMMA 22 guarantees that for $n \geq 0$ and $q \in \mathtt{reach}(n)$ the set $\mathtt{pe}(q, n)$ has the minimal element. We denote it by $\mathtt{pe}(q, n)\Downarrow$. The following property is useful.

Lemma 24 *Let $n \geq 0$. If $q \in reach(n+1)$ then there exists a state $p \in reach(n)$ such that $\mathtt{pe}(q, n+1)\Downarrow = \mathtt{pe}(p, n)\Downarrow a_n\mathtt{tag}(p, q)$.*

PROOF Since $\mathtt{pe}(q, n+1)\downarrow \in \mathtt{pe}(q, n+1)$, there exist $p \in reach(n)$ and $\alpha \in \mathtt{pe}(p, n)$ such that $\mathtt{pe}(q, n+1)\downarrow = \alpha a_n\mathtt{tag}(p, q)$. We show $\alpha = \mathtt{pe}(p, n)\downarrow$. Suppose there exists $\beta \in \mathtt{pe}(p, n)$ such that $\beta \lessdot \alpha$. LEMMA 19 yields $\beta a_n\mathtt{tag}(p, q) \lessdot \alpha a_n\mathtt{tag}(p, q)$. It contradicts with $\mathtt{pe}(q, n+1)\Downarrow = \alpha a_n\mathtt{tag}(p, q)$ because $\beta a_n\mathtt{tag}(p, q) \in \mathtt{pe}(q, n+1)$. Therefore $\alpha = \mathtt{pe}(p, n)\Downarrow$. □

The invariant of main loop of our algorithm is stated as follows.

Definition 10 *For $n \geq 0$ and $P \subseteq Q$, $\mathrm{Inv}(n)$ is a conjunction of the following properties.*

(a) *$p \in \mathtt{reach}(n)$ for any $p \in K$.*

(b) *For any $p \notin K$, if $p \in \mathtt{reach}(n)$ then $\mathtt{pe}(q, n)\Downarrow \lessdot \mathtt{pe}(p, n)\Downarrow$ and the last element of the trace of $\mathtt{pe}(p, n)\Downarrow$ wrt. $\mathtt{pe}(q, n)\Downarrow$ is zero for any $q \in K$.*

(c) *For any $p, q \in K$ with $p \neq q$, $\mathtt{pe}(p, n)\Downarrow \lessdot \mathtt{pe}(q, n)\Downarrow$ iff $D[p][q] = 1$ iff $D[q][p] = -1$.*

(d) *For any $p, q \in K$ with $p \neq q$, $B[p][q]$ is equal to the last element of the trace of $\mathtt{pe}(p, n)\Downarrow$ wrt. $\mathtt{pe}(q, n)\Downarrow$.*

(e) *For any $p \in K$, $P[p][t]$ gives the greatest j such that t is included in α_j for any $t \in T_{\mathrm{cap}}$ that occurs in $\mathtt{pe}(p, n)\downarrow$; $P[p][t] = -1$ for the other $t \in T_{\mathrm{cap}}$, where $\mathtt{pe}(p, n)\Downarrow = \alpha_0 a_1 \alpha_1 \ldots a_n \alpha_n$.*

The key property to show the correctness of our algorithm is that the parenthesis expression reaching for a state with $n+1$ steps, yielded as the result of Algorithm 2, is the least among ones reaching the state with $n+1$ steps.

Lemma 25 *Suppose $\mathrm{Inv}(n)$ holds at line 4 of Algorithm 1. Then the following properties hold when returned from Algorithm 2:*

1. *$\mathtt{pe}(q, n+1)\Downarrow = \mathtt{pe}(p, n)\Downarrow a_n\alpha$ for any $\langle p, q, \alpha \rangle \in trans$,*

2. $trans_1 \subseteq K$, *where* $trans_1 = \{p \mid \langle p, q, \alpha\rangle \in trans\}$, *where* $trans_2 = \{q \mid \langle p, q, \alpha\rangle \in trans\}$.

PROOF Because only the first property is non-trivial, we only show that for every iteration of the outer loop (Lines 2–10) of Algorithm 2 the values of p and q at Line 9 satisfy $\mathtt{pe}(q, n+1)\downarrow = \mathtt{pe}(p, n)\downarrow a_n \mathtt{tag}(p, q)$ for any $n \geq 0$, which immediately concludes that the first property holds at the exit of Algorithm 2. LEMMA 24 implies that $\mathtt{pe}(q, n+1)\Downarrow = \min\{\mathtt{pe}(s, n)\Downarrow a_n \mathtt{tag}(s, q) \mid s \in \mathtt{reach}(n), \mathtt{tag}(s, q) \neq \bot\}$. Thus, we show the right-hand side of this equation should be $\mathtt{pe}(p, n)\Downarrow a_n \mathtt{tag}(p, q)$. For the sake of conciseness, we often refer to $\mathtt{pe}(s, n)\downarrow a_n \mathtt{tag}(s, q)$ as μ_s for any $s \in \mathtt{reach}(n)$ with $\mathtt{tag}(s, q) \neq \bot$.

Let $S = \{s \in K \mid \mathtt{tag}(s, q) \neq \bot\}$. Note that $S \subseteq \mathrm{reach}(n)$ from the property (a) of $\mathrm{Inv}(n)$. We first show that the value of p in Line 9 satisfies $\mu_p \lessdot \mu_s$ for any $s \in S$. Because $\lessdot$ on $\mathtt{pe}(q, n+1)$ is a strict total order from LEMMA 22, it is enough to see that in the line 7 the value of p is updated with that of p' if and only if $\mu_{p'} \lessdot \mu_p$; if it holds, it is easy to see that the **for all** loop in Lines 3–7 computes the minimal parenthesis expression in $\{\mu_s \mid s \in S\}$. Note that $p \neq p'$ in Lines 5. Thus LEMMA 23 impiles $\mathtt{pe}(p, n)\Downarrow \neq \mathtt{pe}(p', n)\Downarrow$. From the property (d) of $\mathrm{Inv}(n)$, the pair $\langle B[p][p'], B[p'][p]\rangle$ is the last element of $\mathtt{tr}(\mathtt{pe}(p, n)\Downarrow; \mathtt{pe}(p', n)\Downarrow)$. Hence $\langle \rho, \rho'\rangle$ obtained in the line 5 is the last element of $\mathtt{tr}(\mu_p; \mu_{p'})$. On the other hand, the property (c) of $\mathrm{Inv}(n)$ shows that $D[p'][p] = 1$ is equivalent to $\mathtt{pe}(p', n)\downarrow \lessdot \mathtt{pe}(p, n)\downarrow$. Suppose the value of p is replaced with that of p'. It follows that the condition of the **if** statement in Line 7 should hold. Then LEMMA 18 immediately yields $\mu_{p'} \lessdot \mu_p$. Suppose $\mu_{p'} \lessdot \mu_p$. Because $\mathtt{pe}(p, n)\Downarrow \neq \mathtt{pe}(p', n)\Downarrow$, LEMMA 18 yields that the **if** statement in Line 7 holds. Therefore the value of p is replaced with that of p' at Line 7.

We next show that the value of p in Line 9 satisfies $\mu_p \lessdot \mu_s$ for $s \in \mathtt{reach}(n)|_q \backslash K$. From $\mathrm{Inv}(n)$ (b) the last element of $\mathtt{tr}(\mathtt{pe}(s, n)\downarrow; \mathtt{pe}(p, n)\downarrow)$ is $\langle 0, 0\rangle$ and $\mathtt{pe}(p, n)\downarrow \lessdot \mathtt{pe}(s, n)\downarrow$, and hence the last element of $\mathtt{tr}(\mu_s; \mu_p)$ is also $\langle 0, 0\rangle$. Therefore we have $\mu_p \lessdot \mu_s$. □

Now we turn to the analysis of the algorithm 3. We denote the value of K at the entry point of Algorithm 3 by K'. $Inv'(n)$ is introduced to denote the similar property to $Inv(n)$ except that the value of K', D', B' and P' are used instead of K, D, B and P respectively.

Lemma 26 *Suppose* $\mathrm{Inv}(n)$ *holds at Line 4 of Algorithm 1. Then the properties (a) and (b) of* $\mathrm{Inv}(n+1)$ *hold when returned from Algorithm 3.*

PROOF By construction of *trans* in Algorithm 2, it is easy to see that the second component of each element in *trans* belongs to $\mathrm{reach}(n+1)$. Thus the property (a) of $\mathrm{Inv}(n+1)$ immediately holds from the line 1 of Algorithm 3.

Next we show that the property (b) of $\mathrm{Inv}(n+1)$ holds at the exit of Algorithm 3. Let q be in $\mathrm{reach}(n+1)\backslash K$ at the exit of Algorithm 3. By LEMMA 24, there

exists a state p such that $\mathtt{pe}(q, n+1)\!\downarrow = \mathtt{pe}(p, n)\!\downarrow a_n \mathtt{tag}(p, q)$. By construction of K in Algorithm 3 and *trans* in Algorithm 2, we have $p \in \mathrm{reach}(n) \backslash K'$. Let $q' \in K$. Then, by construction of K, there exists a state p' such that $\langle p', q', \mathtt{tag}(p', q') \rangle \in$ *trans*. LEMMA 25 yields $\mathtt{pe}(q', n+1)\!\downarrow = \mathtt{pe}(p', n)\!\downarrow a_n \mathtt{tag}(p', q')$. Let $\langle \sigma, \sigma' \rangle$ be the last element of $\mathtt{tr}(\mathtt{pe}(q, n+1)\!\downarrow; \mathtt{pe}(q', n+1)\!\downarrow)$. By construction of *trans*, we have $p' \in K'$. Thus, from the property (b) of $\mathrm{Inv}'(n)$, the trace of $\mathtt{pe}(p, n)\!\downarrow$ wrt. $\mathtt{pe}(p', n)\!\downarrow$ ends in 0. Then, by the definition of trace we obtain $\sigma = 0$ and hence $\sigma \leq \sigma'$. The property (b) of $\mathrm{Inv}'(n)$ also yields $\mathtt{pe}(p', n)\!\downarrow \lessdot \mathtt{pe}(p, n)\!\downarrow$. Therefore, from LEMMA 18 $\mathtt{pe}(q, n+1)\!\Downarrow \lessdot \mathtt{pe}(q', n+1)\!\Downarrow$ holds. □

Lemma 27 *Suppose* $\mathrm{Inv}(n)$ *holds at Line 4 of Algorithm 1. Then the property (d) of* $\mathrm{Inv}(n+1)$ *holds when returned from Algorithm 3.*

PROOF We show that for any iteration of Lines 3–13 in Algorithm 3 the pair of $B[q][q']$ and $B[q'][q]$ is the last element of $\mathtt{tr}(\mathtt{pe}(q, n+1)\!\downarrow; \mathtt{pe}(q', n+1)\!\downarrow)$. From LEMMA 25 we have $\mathtt{pe}(q, n+1)\!\Downarrow = \mathtt{pe}(p, n)\!\Downarrow a_n \alpha$ and $\mathtt{pe}(q', n+1)\!\Downarrow = \mathtt{pe}(p', n)\!\Downarrow a_n \alpha'$ for each iteration. The result is proved as in the proof of LEMMA 25 when $p \neq p'$. If $p = p'$ then $\mathtt{pe}(p, n)\!\downarrow = \mathtt{pe}(p', n)\!\downarrow$. By construction of *trans*, both α and α' are associated with disinct transitions leaving from the same state. Thus LEMMA 16 implies $\alpha \neq \alpha'$. Therefore the result follows from Line 6 of Algorithm 3. □

Lemma 28 *Suppose* $\mathrm{Inv}(n)$ *holds at Line 4 of Algorithm 1. Then the property (c) of* $\mathrm{Inv}(n+1)$ *holds when returned from Algorithm 3.*

PROOF For each iteration of Lines 3–13 in Algorithm 3 we show that $D[q][q'] = 1$ iff $D[q'][q] = -1$ iff $\mathtt{pe}(q, n+1)\!\downarrow \lessdot \mathtt{pe}(q', n+1)\!\downarrow$ and that $D[q'][q] = 1$ iff $D[q][q'] = -1$ iff $\mathtt{pe}(q', n+1)\!\Downarrow \lessdot \mathtt{pe}(q, n+1)\!\Downarrow$.

By definition $\beta \lessdot \gamma$ and $\gamma \lessdot \beta$ never hold at once for any parenthesis expressions β and γ. Thus $D[q][q'] = 1$, $D[q'][q] = -1$ and $\mathtt{pe}(q, n+1)\!\Downarrow \lessdot \mathtt{pe}(q', n+1)\!\Downarrow$ imply that neither $D[q'][q] = 1$, $D[q][q'] = -1$ nor $\mathtt{pe}(q', n+1)\!\downarrow \lessdot \mathtt{pe}(q, n+1)\!\downarrow$ hold, and vice versa. Furthermore, from Line 11 we have $D[q][q'] = i$ iff $D[q'][q] = -i$ where $i = 1$ or $i = -1$. Thus it suffices to show that either $D[q][q'] = 1$ and $\mathtt{pe}(q, n+1)\!\Downarrow \lessdot \mathtt{pe}(q', n+1)\!\Downarrow$, or $D[q][q'] = -1$ and $\mathtt{pe}(q', n+1)\!\Downarrow \lessdot \mathtt{pe}(q, n+1)\!\Downarrow$.

As shown in the previous lemma, the pair of ρ and ρ' is the last element of $\mathtt{tr}(\mathtt{pe}(q, n+1)\!\Downarrow; \mathtt{pe}(q', n+1)\!\Downarrow)$. We distinguish the following three cases according to the relationship between ρ and ρ'.

[1] Suppose $\rho > \rho'$. Then from Line 11 we have $D[q][q'] = 1$. LEMMA 18 yields $\mathtt{pe}(q, n+1)\!\Downarrow \lessdot \mathtt{pe}(q', n+1)\!\Downarrow$.

[2] If $\rho < \rho'$ then, as in the previous case, we have $D[q][q'] = -1$ and $\mathtt{pe}(q', n+1)\!\Downarrow \lessdot \mathtt{pe}(q, n+1)\!\Downarrow$.

[3] Suppose $\rho = \rho'$. From LEMMA 25 we have $\mathtt{pe}(q, n+1)\!\downarrow = \mathtt{pe}(p, n)\!\downarrow a_n \alpha$ and $\mathtt{pe}(q', n+1)\!\downarrow = \mathtt{pe}(p', n)\!\downarrow a_n \alpha'$ for p, p', α and α' in the iteration. Suppose

$p = p'$. Then $\mathtt{pe}(p,n)\downarrow = \mathtt{pe}(p',n)\downarrow$ and that both α and α' are associated with disinct transitions leaving from the same state. Thus, from LEMMA 16, either $\alpha \sqsubset' \alpha'$ or $\alpha' \sqsubset' \alpha$ holds. If the former holds, $D[q][q'] = 1$ and, from Lemma 18, $\mathtt{pe}(q,n)\downarrow \lessdot \mathtt{pe}(q',n)\downarrow$. Otherwise, $D[q][q'] = -1$ and $\mathtt{pe}(q',n)\downarrow \lessdot \mathtt{pe}(q,n)\downarrow$. Suppose $p \neq p'$. From LEMMA 23 one of the properties $\mathtt{pe}(p,n)\Downarrow \lessdot \mathtt{pe}(p',n)\Downarrow$ or $\mathtt{pe}(p',n)\Downarrow \lessdot \mathtt{pe}(p,n)\Downarrow$ holds. By assumption $\mathrm{Inv}(n)$ holds and so does $\mathrm{Inv}'(n)$. Hence the property (c) of $\mathrm{Inv}'(n)$ yields either $D[p][p'] = 1$ or $D[p][p'] = -1$. If the former holds, $D[q][q'] = 1$ and, from Lemma 18, $\mathtt{pe}(q,n)\Downarrow \lessdot \mathtt{pe}(q',n)\Downarrow$. Otherwise, $D[q][q'] = -1$ and $\mathtt{pe}(q',n)\Downarrow \lessdot \mathtt{pe}(q,n)\Downarrow$. □