Computing ϵ-Free NFA from Regular Expressions in $O(n \log^2(n))$ Time*

Christian Hagenah and Anca Muscholl

Institut für Informatik, Universität Stuttgart,
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany

Abstract. The standard procedure to transform a regular expression to an ϵ-free NFA yields a quadratic blow-up of the number of transitions. For a long time this was viewed as an unavoidable fact. Recently Hromkovič et.al. [5] exhibited a construction yielding ϵ-free NFA with $O(n \log^2(n))$ transitions. A rough estimation of the time needed for their construction shows a cubic time bound. The known lower bound is $\Omega(n \log(n))$. In this paper we present a sequential algorithm for the construction described in [5] which works in time $O(n \log(n) + \text{size of the output})$. On a CREW PRAM the construction is possible in time $O(\log(n))$ using $O(n + (\text{size of the output})/\log(n))$ processors.

1 Introduction

Among various descriptions of regular languages regular expressions are especially interesting because of their succinctness. On the other hand, the high degree of expressiveness leads to algorithmically hard problems, for example testing equivalence is PSPACE-complete. Given a regular expression we are often interested in computing an equivalent nondeterministic finite automaton *without ϵ-transitions (NFA)*. This conversion is of interest due to some operations which can be easily performed on NFA, as for example intersection.

In this paper we present efficient sequential and parallel algorithms for converting regular expressions into small NFA. For a regular expression E we take the number of letters as the size of E, whereas the size of an NFA is measured as the number of transitions. It is known that the translation from NFA to regular expressions can yield an exponential blow-up, [3]. The other direction however can be achieved in polynomial time. One classical method for constructing NFA from regular expressions is based on position automata. This construction yields NFA of quadratic size, see e.g. [1, 2]. A substantial improvement on this construction was achieved in [5], where a refinement of position automata was shown to yield NFA with $O(n \log^2(n))$ transitions. This is optimal up to a possible $\log(n)$ factor, as shown in [5] by proving a $O(n \log(n))$ lower bound. However, the precise complexity of the conversion proposed in [5] was not investigated. A trivial estimation of the construction of [5] leads to a cubic algorithm.

* Research was partly supported by the French-German project PROCOPE.

Performing the conversion form regular expressions to NFA efficiently is important from a practical viewpoint. The best one can hope for is to perform the construction in time proportional to the output size. In the present paper we propose efficient sequential and parallel algorithms for converting regular expressions to NFA. Our approach is based on the construction proposed in [5], but using a slightly different presentation. This allows us to obtain an algorithm which works in time $O(n \log(n) + \text{size of the output})$. Therefore, our algorithm has worst-case complexity of $O(n \log^2(n))$. In the parallel setting we are able to perform the construction on a CREW PRAM in $O(\log(n))$ time by using $O(n)$ processors for computing the description of the states of the NFA, resp. $O(n \log(n))$ processors in the worst-case for the output NFA. Previously known was an $O(\log(n))$ time algorithm using $O(n/\log(n))$ processors, which computes an NFA *with* ϵ-transitions, see [4]. The paper is organized as follows. The sequential algorithm is presented in Sect. 4. Basic notions on position automata are recalled in Sect. 2, whereas Sect. 3 deals with the common follow sets construction of [5].

2 Preliminaries

Let A denote a finite alphabet. We consider non-empty regular expressions over A, i.e. (bracketed) expressions E built from ϵ and the letters in A, using concatenation $\cdot$, union $+$ and Kleene star $*$. The regular language defined by a regular expression E is denoted $\mathcal{L}(E)$. Finite automata are denoted as usual as $\mathcal{A} = (Q, A, q_0, \delta, F)$, with Q as set of states, $\delta \subseteq Q \times A \times Q$ as transition relation, q_0 as initial state and F as set of final states. The language recognized by $\mathcal{A}$ is denoted $L(\mathcal{A})$.

For algorithmic purposes a regular expression E over A is given by some syntax tree t_E. The syntax tree t_E has leaves labelled by ϵ or $a \in A$, and the inner nodes are either binary and labelled by $+$ or $\cdot$, or they are unary and labelled by $*$. The inner nodes of a syntax tree will be named $F, G, \ldots$ and we will denote them as subexpressions of E. For two subexpressions F, G of E we write $F \leq G$ ($F < G$, resp.) if F is an ancestor (a proper ancestor, resp.) of G. For a subexpression F let firststar(F) denote the largest subexpression G with $G \leq F$ such that G^* is the parent node of G.

A subtree t of t_E is a connected subgraph (i.e. a tree) of t_E. A subtree t is called *full subtree* if it contains all descendants of its root. This means that a full subtree of t_E corresponds to a subexpression of E.

We may suppose without loss of generality that the leaves of t_E are labelled with pairwise distinct letters. This allows to identify the leaves of t_E labelled by A uniquely by their labelling. For example, for $E = (a^* + b)^* ab^*$ we replace A by $\{a_1, a_2, b_1, b_2\}$ and E by $(a_1^* + b_1)^* a_2 b_2^*$.

3 Position Automata

In this section we recall some basic notions related to the construction of position automata from regular expressions. We follow the definitions from [2, 5].

3.1 Positions and sets of positions

Given a regular expression E, the set $\mathrm{pos}(E)$ comprises all positions of E which are labelled by letters from A. According to our convention, $\mathrm{pos}(E) \subseteq A$. Positions of E will be named $x, y, \ldots$.

Lemma 1. *Let E be a regular expression, $n = |pos(E)|$. Then we can compute in linear time an equivalent expression E', $\mathcal{L}(E) = \mathcal{L}(E')$, such that E' has length $O(n)$.*

The size $|E|$ of the expression E is defined as $|\mathrm{pos}(E)|$. Moreover, $\mathrm{pos}(t)$ and $|t|$ are defined analogously for a subtree t of t_E. Throughout the paper we denote by n the size $|E|$ of E. The lemma above says that we may assume that the size of the syntax tree t_E satisfies $|t_E| \in O(n) = O(|\mathrm{pos}(t_E)|)$.

For a regular expression E we consider two distinguished subsets of positions, $\mathrm{first}(E)$ and $\mathrm{last}(E)$. The set $\mathrm{first}(E) \subseteq \mathrm{pos}(E)$ contains all positions which can occur as first letter in some word in $\mathcal{L}(E)$. Similarly, $\mathrm{last}(E)$ contains all positions which can occur as last letter in some word in $\mathcal{L}(E)$. Formally:

$$\mathrm{first}(E) = \{x \in \mathrm{pos}(E) \mid xA^* \cap \mathcal{L}(E) \neq \emptyset\},$$
$$\mathrm{last}(E) = \{x \in \mathrm{pos}(E) \mid A^*x \cap \mathcal{L}(E) \neq \emptyset\}.$$

The sets $\mathrm{first}(E), \mathrm{last}(E)$ can be computed inductively by noting that e.g. $\mathrm{first}(F+G) = \mathrm{first}(F) \cup \mathrm{first}(G)$, $\mathrm{first}(F^*) = \mathrm{first}(F)$ and $\mathrm{first}(F \cdot G) = \mathrm{first}(F)$ if $\epsilon \notin \mathcal{L}(F)$, resp. $\mathrm{first}(F \cdot G) = \mathrm{first}(F) \cup \mathrm{first}(G)$ if $\epsilon \in \mathcal{L}(F)$. For a given position $x \in \mathrm{pos}(E)$ let $\mathrm{follow}(x) \subseteq \mathrm{pos}(E)$ contain all positions y which are immediate successors of x in some word of $\mathcal{L}(E)$:

$$\mathrm{follow}(x) = \{y \in \mathrm{pos}(E) \mid A^*xyA^* \cap \mathcal{L}(E) \neq \emptyset\}.$$

As above, $\mathrm{follow}(x)$ can be defined recursively by means of $\mathrm{follow}(x, F) = \mathrm{follow}(x) \cap \mathrm{pos}(F)$. We omit the definition here, since anyway we will not compute the sets $\mathrm{follow}(x)$ globally.

3.2 Automata

First, last and follow sets are the basic components of an NFA $\mathcal{A}_E$ recognizing $\mathcal{L}(E)$, called *position automaton* in [5]. Let $\mathcal{A}_E = (Q, A, \delta, q_0, F)$ be defined by

$$Q = \mathrm{pos}(E) \dot\cup \{q_0\}$$
$$\delta = \{(q_0, x, x) \mid x \in \mathrm{first}(E)\} \cup \{(x, y, y) \mid y \in \mathrm{follow}(x)\}$$
$$F = \begin{cases} \mathrm{last}(E) & \text{if } \epsilon \notin \mathcal{L}(E) \\ \mathrm{last}(E) \cup \{q_0\} & \text{otherwise} \end{cases}$$

Recall for the above definition that $\mathrm{pos}(E) \subseteq A$. The following equivalence is easy to check:

Proposition 2. *For every regular expression E we have $\mathcal{L}(\mathcal{A}_E) = \mathcal{L}(E)$.*

The construction above yields ϵ-free automata with $n+1$ states and $O(n^2)$ transitions. In [5] a refined construction was presented, based on the idea of a *system of common follow sets (CFS system)*, which is defined as follows:

Definition 3 ([5]). *Let E be a regular expression. A CFS system S for E is given as $S = (dec(x))_{x \in pos(E)}$, where $dec(x) \subseteq \mathcal{P}(pos(E))$ is a decomposition of $follow(x)$:*

$$follow(x) = \bigcup_{C \in dec(x)} C\,.$$

Let $\mathcal{C}_S = \{first(E)\} \cup \bigcup_{x \in pos(E)} dec(x)$. The CFS automaton $\mathcal{A}_S$ associated with S is defined as $\mathcal{A}_S = (Q, A, q_0, \delta, F)$ where

$$\begin{aligned} Q &= \mathcal{C}_S \times \{0,1\} \\ q_0 &= \begin{cases} (first(E),1) & \text{if } \epsilon \in \mathcal{L}(E) \\ (first(E),0) & \text{otherwise} \end{cases} \\ \delta &= \{(C,f), x, (C',f')) \mid x \in C,\ C' \in dec(x) \text{ and } f' = 1 \Leftrightarrow x \in last(E)\} \\ F &= \mathcal{C}_S \times \{1\} \end{aligned}$$

Lemma 4. *Let E be a regular expression and let S be a CFS system for E. Then the CFS automaton $\mathcal{A}_S$ recognizes $\mathcal{L}(E)$.*

It is shown in [5] how to obtain a CFS system S for a given regular expression E such that $|\mathcal{C}_S| \in O(n)$, $\sum_{C \in \mathcal{C}_S} |C| \in O(n \log n)$ and $|\mathrm{dec}(x)| \in O(\log n)$ for all $x \in \mathrm{pos}(E)$. This yields a CFS automaton with $O(n)$ states and $O(n \log^2(n))$ transitions.

4 Computing a Common Follow Sets System

4.1 Properties of follow sets

The running time of our algorithm relies heavily on some structural properties of follow sets which are discussed in the following.

Lemma 5. *Let E be a regular expression and let F, G be subexpressions with $E \leq F \leq G$. Then we have:*

1. *$first(F) \cap first(G) \neq \emptyset$ implies $first(G) \subseteq first(F)$.*
2. *$F \leq H \leq G$ and $\emptyset \neq first(G) \subseteq first(F)$ implies $first(G) \subseteq first(H) \subseteq first(F)$.*
3. *$x \in pos(G) \setminus first(G)$ implies $x \notin first(F)$.*

The proof of the lemma is a straightforward application of the inductive definition. An analogous lemma can be also stated for last sets.

The next lemma deals with the relation between follow sets and a decomposition of the syntax tree, which will be used recursively in the definition of the CFS system. For simplifying the notation we will denote for $x \in \mathrm{pos}(E)$, $E \leq F$, the set $\mathrm{follow}(x) \cap \mathrm{pos}(F)$ by $\mathrm{follow}_F(x)$. Analogously, $\mathrm{follow}_t(x)$ denotes the set $\mathrm{follow}(x) \cap \mathrm{pos}(t)$ for a subtree t.

Lemma 6. *Given a regular expression E, a syntax tree t_E and subexpressions F, G with $E < F < G$. Let t, t' be subtrees of t_E such that $pos(t) \subseteq pos(F) \setminus pos(G)$ and $pos(t') \subseteq pos(G)$. Then we have for all $x, x', y \in pos(E)$:*

1. *$follow_F(x) = \emptyset$ for all $x \in pos(t') \setminus last(G)$;*
2. *$follow_F(x) = follow_F(x')$ for all $x, x' \in pos(t') \cap last(G)$;*
3. *$follow_{t'}(y) = first(G) \cap pos(t')$ for all $y \in pos(t)$ with $follow_{t'}(y) \neq \emptyset$.*

4.2 Recursive definition of CFS systems

The CFS system defined in [5] is based on a divide-and-conquer construction. Consider a subtree t of t_E and let F denote the root of t. Let $x \in \mathrm{pos}(t)$. If $|t| = 1$ then we define

$$C_0 = \mathrm{follow}_t(x) = \mathrm{follow}(x) \cap \{x\}, \quad \mathrm{dec}(x,t) = \{C_0\}\,.$$

Suppose now that $|t| > 1$. Then let t_1 be a subtree of t such that $1/3|t| \leq |t_1| \leq 2/3|t|$ and let $t_2 = t \setminus t_1$. Let F_1 denote the root of t_1. Clearly, for every position $x \in \mathrm{pos}(t)$ we have $\mathrm{follow}_t(x) = \mathrm{follow}_{t_1}(x) \,\dot\cup\, \mathrm{follow}_{t_2}(x)$. We distinguish two cases, depending on $x \in \mathrm{pos}(t_1)$ or $x \in \mathrm{pos}(t_2)$.

i) Let $x \in \mathrm{pos}(t_1)$. If $x \notin \mathrm{last}(F_1)$ then by Lem. 6 we have $\mathrm{follow}_{t_2}(x) = \emptyset$. Otherwise, for $x \in \mathrm{last}(F_1)$ then again by Lem. 6 we have $\mathrm{follow}_{t_2}(x) = \mathrm{follow}_{t_2}(x')$ for all $x' \in \mathrm{last}(F_1) \cap \mathrm{pos}(t_1)$.
Let $C_1 = \mathrm{follow}_{t_2}(x')$ for some $x' \in \mathrm{pos}(t_1) \cap \mathrm{last}(F_1)$ and define $\mathrm{dec}(x,t)$ as

$$\mathrm{dec}(x,t) = \begin{cases} \mathrm{dec}(x,t_1) & \text{if } x \notin \mathrm{last}(F_1) \\ \mathrm{dec}(x,t_1) \cup \{C_1\} & \text{otherwise} \end{cases}$$

ii) Let $x \in \mathrm{pos}(t_2)$. If $\mathrm{follow}_{t_1}(x) \neq \emptyset$ then we have $\mathrm{follow}_{t_1}(x) = \mathrm{first}(F_1) \cap \mathrm{pos}(t_1)$ by Lem. 6.
Let $C_2 = \mathrm{first}(F_1) \cap \mathrm{pos}(t_1)$ and define $\mathrm{dec}(x,t)$ as

$$\mathrm{dec}(x,t) = \begin{cases} \mathrm{dec}(x,t_2) & \text{if } \mathrm{follow}_{t_1}(x) = \emptyset \\ \mathrm{dec}(x,t_2) \cup \{C_2\} & \text{otherwise} \end{cases}$$

It can be easily verified that $\mathrm{dec}(x,t)$ is a decomposition[1] of $\mathrm{follow}_t(x)$, i.e. $\mathrm{follow}_t(x) = \bigcup_{C \in \mathrm{dec}(x,t)} C$. Hence, we obtain a CFS system $\mathcal{C}(t)$ restricted to t, where

$$\mathcal{C}(t) = \bigcup \{\mathrm{dec}(x,t) \mid x \in \mathrm{pos}(t)\} = \{C \mid C \in \mathrm{dec}(x,t) \text{ for some } x \in \mathrm{pos}(t)\}\,.$$

Note that $|\mathcal{C}(t)| \leq |\mathcal{C}(t_1)| + |\mathcal{C}(t_2)| + 2$. This yields $|\mathcal{C}(t)| \leq 3|t| - 2$. Similarly, the following estimations can be easily verified (see also Lem. 4 of [5]):

$$\begin{aligned} &\textstyle\sum_{C \in \mathcal{C}(t)} |C| \leq 2|t| \log(|t|) + 1 \text{ and} \\ &|\mathrm{dec}(x,t)| \leq 2\log(|t|) + 1, \text{ for all } x \in \mathrm{pos}(t)\,. \end{aligned}$$

[1] In [5] the corresponding set $\bigcup_{C \in \mathrm{dec}(x,t)} C$ is just a subset of $\mathrm{follow}_t(x)$. Having equality here simplifies the recursive definition and the correctness proof of the decomposition.

5 A Sequential $O(n \log(n))$ Algorithm for Computing a Common Follow Sets System

We consider now the computation of the sets defined in the previous section. For $C_0 = \text{follow}(x) \cap \{x\}$ we can determine whether $x \in \text{follow}(x)$ by checking whether $x \in \text{last}(S) \cap \text{first}(S)$ for $S = \text{firststar}(x)$. For the recursion step we have to determine C_1, C_2 with

$$C_1 = \text{follow}_{t_2}(x) \quad \text{and} \quad C_2 = \text{first}(F_1) \cap \text{pos}(t_1)\,.$$

We want to compute both C_1, C_2 and the positions $x \in \text{pos}(t)$ to which C_1 or C_2 is added in linear time, i.e. in time $O(|t|)$. As shown below, the computation of C_1 reduces to computing a union of first sets restricted to $\text{pos}(t_2)$. This yields two problems: First we need an efficient way to compute intersections of first sets with a given set of positions. Second, the union of restricted first sets has to be disjoint. The solution to both problems will rely on a suitable data structure for first sets. Before discussing the data structure let us consider the set C_1 in more detail[2].

Definition 7. *Let $E \leq F$ be regular expressions. We define $\mathit{fnext}(F) \subseteq \mathit{pos}(E)$ as*

$$\mathit{fnext}(F) = \begin{cases} \mathit{first}(G) & \text{if } F \cdot G \text{ is the parent node of } F \\ \mathit{first}(F) & \text{if } F^* \text{ is the parent node of } F \\ \emptyset & \text{otherwise} \end{cases}$$

Analogously, $\mathit{lprev}(F)$ is defined by replacing first by last and by requiring that $G \cdot F$ is the parent node of F.

Using the fnext operator we are able to express follow sets as unions of first sets. Compared with Lem. 3 in [5] we need for expressing $\text{follow}_{t_2}(x)$ at most *one* first set which is not contained in F. Of course, this is necessary in order to be able to determine C_1 in time $O(|t|)$:

Proposition 8. *Let E be a regular expression with $E \leq F < F_1$ and let t_E be a syntax tree of E. Let t_2 be a subtree with root F and $\mathit{pos}(t_2) \cap \mathit{pos}(F_1) = \emptyset$ and consider a position $x \in \mathit{last}(F_1)$. Then we have*

$$\mathit{follow}_{t_2}(x) = \bigcup_{G \in \mathcal{G}} (\mathit{fnext}(G) \cap \mathit{pos}(t_2))$$

where the union is taken over

$$\mathcal{G} = \{G \mid \mathit{last}(G) \supseteq \mathit{last}(F_1) \text{ and } (F < G \leq F_1 \text{ or } G = \mathit{firststar}(F))\}\,.$$

[2] In the definition below fnext(F) corresponds to first(fnext(F)) in [5].

Proof: Note that for every $G \leq F_1$ with $\mathrm{last}(F_1) \subseteq \mathrm{last}(G)$ we have $\mathrm{fnext}(G) \cap \mathrm{pos}(t_2) \subseteq \mathrm{follow}_{t_2}(x)$. Conversely, consider a position $y \in \mathrm{follow}_{t_2}(x)$ with $y \notin \mathrm{fnext}(G)$, for all $F < G \leq F_1$ with $\mathrm{last}(F_1) \subseteq \mathrm{last}(G)$. Hence, there exists some node G, $E \leq G \leq F$, with $y \in \mathrm{fnext}(G)$ and $\mathrm{last}(F_1) \subseteq \mathrm{last}(G)$. Clearly, the parent node of G is G^* (otherwise, $\mathrm{fnext}(G) \cap \mathrm{pos}(t_2) = \emptyset$), thus $y \in \mathrm{first}(G) \cap \mathrm{pos}(t_2)$. If $G = \mathrm{firststar}(F)$ then we are done. Otherwise $G < H = \mathrm{firststar}(F)$. In this case it is not difficult to verify using Lem. 5 that for all $G < H$ with $\mathrm{first}(G) \cap \mathrm{pos}(t_2) \neq \emptyset$ we also have $\mathrm{first}(G) \cap \mathrm{pos}(t_2) = \mathrm{first}(H) \cap \mathrm{pos}(t_2)$. Therefore, $y \in \mathrm{first}(H) \cap \mathrm{pos}(t_2)$. □

Our algorithm is based on a suitable order on positions of E, which allows manipulating first sets efficiently. We use an array called $\mathsf{firstdata}$ such that for each subexpression F of E the set $\mathrm{first}(F)$ is a subinterval of $\mathsf{firstdata}$.The crucial point is the order of positions within $\mathsf{firstdata}$. Consider a fixed syntax tree t_E of E. We first define a forest $\mathcal{F}$ by deleting all edges from nodes labelled $F \cdot G$ to the child labelled G, whenever $\epsilon \notin \mathcal{L}(F)$. Let $\mathcal{F} = \{T_1, \ldots, T_k\}$ be the forest thus obtained, then we denote the trees T_i as *first-trees*. Note that each $\mathrm{first}(F)$ is the union of all $\mathrm{first}(F')$ with $F < F'$ where F' belongs to the same first-tree as F.

We define a total order on $\mathcal{F}$ as follows. For $1 \leq i \neq j \leq k$ let $T_i \prec T_j$ whenever the roots F_i, F_j of T_i, resp. T_j satisfy

- either $F_j < F_i$, i.e. F_j is an ancestor of F_i,
- or F_i and F_j are incomparable w.r.t. $<$ and F_i lies to the right of F_j.

The order $\prec$ corresponds thus to a reversed preorder traversal of t_E, i.e. right child—left child—parent node.

Suppose that after renaming $\mathcal{F} = \{T_1, \ldots, T_k\}$ with $T_i \prec T_j$ for all $i < j$. The array $\mathsf{firstdata}$ is given as $\mathsf{fdata}(T_1) \cdots \mathsf{fdata}(T_k)$, with $\mathsf{fdata}(T_i)$ being the list of positions corresponding to the yield of T_i. Moreover, by a preorder traversal of each T_i we can determine for each subexpression F of T_i the subinterval of $\mathsf{fdata}(T_i)$ corresponding to $\mathrm{first}(F)$. The set $\mathrm{first}(F)$ is described by its starting position $\mathsf{fstart}(F)$ within $\mathsf{fdata}(T_i)$ and its length $\mathsf{flength}(F) = |\mathrm{first}(F)|$.

Remark 9. (*i*) Let F, G be subexpressions of E. Then we have $\emptyset \neq \mathrm{first}(F) \subseteq \mathrm{first}(G)$ if and only if $\mathsf{fstart}(G) \leq \mathsf{fstart}(F)$ and $\mathsf{fstart}(G)+\mathsf{flength}(G) \geq \mathsf{fstart}(F)+\mathsf{flength}(F)$, i.e. if the subinterval corresponding to $\mathrm{first}(G)$ includes the subinterval corresponding to $\mathrm{first}(F)$. Moreover, $\mathsf{firstdata}$ allows to determine the intersection $\mathrm{first}(F) \cap \mathrm{pos}(t)$ in $O(|t|)$ time, where F is a subexpression and t is subtree of t_E (described as set of positions in increasing order).

(*ii*) A similar data structure $\mathsf{lastdata}$ can be defined for the last sets.

We are now ready to describe an algorithm UnionFirst for the following problem. Given subexpressions F, F_1 of E with $F < F_1$ and subtrees t, t_2 of F, resp. a subtree t_1 of F_1, where $t = t_1 \dot{\cup} t_2$, $\mathrm{pos}(F_1) \cap \mathrm{pos}(t_2) = \emptyset$, and a position $x \in \mathrm{last}(t_1)$. We want to compute the set $C = \mathrm{follow}_{t_2}(x)$. Recall from Prop. 8 that

$$C = \bigcup_{G \in \mathcal{G}} (\mathrm{fnext}(G) \cap \mathrm{pos}(t_2)) \,,$$

with $G \in \mathcal{G}$ if and only if $\text{last}(F_1) \subseteq \text{last}(G)$, and either $F < G \leq F_1$ or $G = \text{firststar}(F)$.

function UnionFirst (node F_1, tree t_2) : nodelist;
  var rootlist, tocheck: nodelist; G: node;
  begin
  rootlist := nil;
  tocheck := nil;
  $G := F_1$;
  while ($G \neq$ root(t_2) and $\text{last}(F_1) \subseteq \text{last}(G)$) **do**
    begin
    $A :=$ parent expression of G;
    if $A = G^*$ **then**
      rootlist := rootlist $\setminus \{H \mid H \in$ tocheck and $\text{first}(H) \subseteq \text{first}(G)\}$;
      rootlist := rootlist $\circ G$;
      tocheck := $\{G\}$;
    else if $A = G \cdot H$ **then**
        if $\epsilon \in \mathcal{L}(G)$ **then** rootlist := rootlist $\circ H$;
        else rootlist := $H\circ$ rootlist **endif**;
        tocheck := tocheck $\cup \{H\}$;
    endif;
    $G := A$;
  endwhile;
  return(rootlist);
  end

The proof of the next proposition is omitted for lack of space.

Proposition 10. *Let F, F_1 be subexpressions of E, $E \leq F < F_1$, and let t_E be a syntax tree. Let t_1 be a subtree with root F_1 and let t_2 be a subtree with root F and $pos(F_1) \cap pos(t_2) = \emptyset$. Let $x \in last(t_1)$ be a position and let $\mathcal{G}$ be defined as above. Then UnionFirst(F_1, t_2) yields a list* rootlist $= (H_1, \ldots, H_l)$ *of (names of) subexpressions of E satisfying the following:*

1. $\bigcup_{i=1}^{l} first(H_i) = \bigcup_{G \in \mathcal{G}, G \neq firststar(F)} fnext(G)$.
   Moreover, $first(H_i) \cap first(H_j) = \emptyset$ for all $i \neq j$.
2. *Let $T(H_i)$ denote the first-tree in the forest $\mathcal{F}$ containing H_i. Then $T(H_i) \preceq T(H_j)$ for all $1 \leq i < j \leq k$. Moreover, if $T(H_i) = T(H_j)$ then H_i precedes H_j w.r.t. preorder (in t_E).*
3. *UnionFirst(F_1, t_2) runs in $O(|t_2|)$ steps.*

Remark 11. Given the assertion of Prop. 10 it is not hard to verify that the set $\bigcup_{G \in \mathcal{G}}(\text{fnext}(G) \cap \text{pos}(t_2))$ can be computed in $O(|t_2|)$ steps using rootlist and firstdata. More precisely, we can precompute in time $O(|t_2|)$ a list fdata(t_2) corresponding to $\text{pos}(t_2)$ sorted as firstdata. Next, we scan rootlist and fdata(t_2) in parallel, building the intersection. Hereby we use Rem. 9 in order to determine in constant time whether a position belongs to a set $\text{first}(G)$. Note that rootlist

has at most $|t_2|$ elements, since the while loop in UnionFirst is executed at most $|t_2|$ times. Finally, if $S = \text{firststar}(F)$ is defined we can check whether $\text{last}(F_1) \subseteq \text{last}(S)$ in $O(1)$ time and compute $\text{first}(S) \cap \text{pos}(t_2)$ in time $O(|t_2|)$.

Theorem 12. *Given a regular expression E and a syntax tree t_E for E of size $O(|E|) = O(n)$. We can compute a CFS system S for E in time $O(n \log(n))$. Therefore, we can compute an NFA $\mathcal{A}_S$ for E of size $|\mathcal{A}_S|$ in time $O(n \log(n) + |\mathcal{A}_S|)$. The worst-case complexity of the algorithm is thus $O(n \log^2(n))$.*

Proof: Recall the recursive definition of $\text{dec}(x, t)$ given in Sect. 4.2. For a position $x \in \text{pos}(t_1)$ we test whether $x \in \text{last}(F_1)$ in constant time (using lastdata), whereas $C_1 = \text{follow}_{t_2}(x)$ can be computed in time $O(|t_2|)$. The case where $x \in \text{pos}(t_2)$ is dual. Here, the set $C_2 = \text{first}(F_1) \cap \text{pos}(t_1)$ can be determined in constant time, whereas determining which $x \in \text{pos}(t_2)$ satisfy $\text{follow}_{t_1}(x) \neq \emptyset$ requires $O(|t_1|)$ steps. To see this, note that for a position $y \in C_2$ we have $\{x \in \text{pos}(t_2) \mid \text{follow}_{t_1}(x) \neq \emptyset\} = \text{precede}(y) \cap \text{pos}(t_2)$, where $\text{precede}(y)$ is defined as the dual of $\text{follow}(y)$, i.e., $\text{precede}(y) = \{x \mid A^*xyA^* \cap \mathcal{L}(E) \neq \emptyset\}$. Moreover, by duality we have $\text{precede}(y) \cap \text{pos}(t_2) = \bigcup_{G \in \mathcal{G}} (\text{lprev}(G) \cap \text{pos}(t_2))$, with $G \in \mathcal{G}$ if $\text{first}(F_1) \subseteq \text{first}(G)$, and either $F < G \leq F_1$ or $G = \text{firststar}(F)$.

Therefore, we can compute in time $O(|t|)$ the sets $\text{dec}(x, t)$ from $\text{dec}(x, t_1)$ and $\text{dec}(x, t_2)$ for all positions x of t. Hence, our algorithm runs in time $O(n \log(n))$. Finally, outputting the transitions of the NFA $\mathcal{A}_S$ is possible in time $O(|\mathcal{A}_S|)$. □

In the parallel setting we have again an output-size optimal algorithm on a CREW PRAM, as stated below. For lack of space we omit the proofs.

Theorem 13. *Given a regular expression E and a syntax tree t_E for E of size $O(|E|) = O(n)$. We can compute a CFS system S for E on a CREW PRAM in time $O(\log(n))$ using $O(n)$ processors. Therefore, we can compute an NFA $\mathcal{A}_S$ for E of size $|\mathcal{A}_S|$ in time $O(\log(n))$ using $O(n + |\mathcal{A}_S|/\log(n))$ processors (i.e., $O(n \log(n))$ processors in the worst case).*

Acknowledgment: We thank Volker Diekert for many comments and contributions and to the anonymous referees for suggestions which helped improving the presentation.

References

1. G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.
2. A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:197–213, 1993.
3. A. Ehrenfeucht and P. Zeiger. Complexity measures for regular expressions. *Journal of Computer and System Sciences*, 12:134–146, 1976.
4. A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1989.
5. J. Hromkovič, S. Seibert, and T. Wilke. Translating regular expressions into small ϵ-free nondeterministic finite automata. In *Proc. of the 14th Ann. Symp. on Theor. Aspects of Comp. Sci. (STACS'97)*, no. 1200 in LNCS, p. 55–66, 1997. Springer.