Regular Expressions with Numerical Constraints and Automata with Counters

Dag Hovland `dag.hovland@uib.no`

Department of Informatics, University of Bergen, Norway

Abstract. Regular expressions with numerical constraints are an extension of regular expressions, allowing to bound numerically the number of times that a subexpression should be matched. Expressions in this extension describe the same languages as the usual regular expressions, but are exponentially more succinct.
We define a class of *finite automata with counters* and a *deterministic* subclass of these. Deterministic finite automata with counters can recognize words in linear time. Furthermore, we describe a subclass of the regular expressions with numerical constraints, a polynomial-time test for this subclass, and a polynomial-time construction of deterministic finite automata with counters from expressions in the subclass.

1 Introduction

Regular expressions with numerical constraints add the possibility to express that a subexpression must be matched a number of times specified by a lower and a upper limit. The Single UNIX Specification [1] requires this as a standard part of regular expressions. In the GNU version of the UNIX program grep [2] and in the programming language Perl they are included as standard and in XML Schemas [3] the 1-unambiguous subclass is allowed. In GNU grep you can, for example, write `([0-9]{1,3}\.){3}[0-9]{1,3}` to match any IPv4 address in dotted-decimal notation.

Common uses of regular expressions with numerical constraints are matching and searching. With matching we mean the problem of deciding whether a given word is in the language defined by the regular expression. Searching means to decide whether one or more of the sub-strings of a given text match the regular expression. Kilpeläinen and Tuhkanen [4] showed that for the regular expressions with numerical constraints, matching can be done with a dynamic programming algorithm in quadratic space and time, relative to the size of the word being matched. Using this algorithm, one can also search in polynomial time.

However, many programs that search using regular expressions with numerical constraints use algorithms with super-polynomial behaviour in the size of the regular expression. These programs typically have as input one short regular expression and many, long, texts to be searched. It is therefore common to construct a deterministic finite automaton (DFA) for matching or searching, as a DFA can be used to search in time linear in the length of the text, although

a quadratic algorithm is usually preferred, as it is faster in most practical cases. The known algorithms for constructing a DFA from a given regular expression with numerical constraints use super-polynomial space.

As an example, consider an experiment lasting 100 hours, where we need to record the moments at which some (unspecified) events take place. We will use one string to describe each 100-hour experiment. For each hour when there is an event, the hour is given, followed by "`h`", followed by a string describing the events occurring that hour. This string is formatted in the following way: for each minute when there is an event, the minute is given, followed by "`m`", followed by the second and "`s`" for each second at which there was an event during that minute. If there were, e.g., a total of three events during one experiment, at 3:12:22, 3:12:43 and 20:45:01, then the string describing the experiment is `3h12m22s43s20h45m1s`. For testing the strings we decide to use the regular expression $((0+\cdots+9)^{1..2}h((1+\cdots+5)^{0..1}(0+\cdots+9)m((1+\cdots+5)^{0..1}(0+\cdots+9)s)^{1..60})^{1..60})^{0..100}$ by executing the command in Fig. 1 (See next section for syntax and semantics of the regular expressions). However, this command turns out to use over 2 gigabytes of memory[1], independent of the length of the text.

```
grep -E "([0-9]{1,2}h([1-5]?[0-9]m([1-5]?[0-9]s){1,60}){1,60}){0,100}"
```

Fig. 1. Example execution of grep

An algorithm for the matching problem will be called a *fast-matcher*, if there is a constant c such that the algorithm runs in time $O(|r|^c \cdot |w|)$ (where r is the regular expression and w is the word to be matched). There exists a fast-matcher for the usual regular expressions without numerical constraints. The algorithm constructs a non-deterministic finite automaton (NFA) recognizing the regular expression, and runs the NFA on the word by maintaining the set of reachable states. The latter set is limited by the size of the NFA, and the number of steps is exactly the length of the word. Construction of an NFA recognizing a regular expression is possible in polynomial time. Brüggemann-Klein [5] describes a different fast-matcher for a subset of the regular expressions, called 1-unambiguous regular expressions. Their algorithm constructs in polynomial time a deterministic finite automaton from a 1-unambiguous regular expression. However, no polynomial-time construction is known for 1-unambiguous regular expressions with numerical constraints.

In this article we describe *finite automata with counters*, and a fast-matcher for a subset of the regular expressions with numerical constraints, called *counter-1-unambiguous regular expressions*. The algorithm works by constructing deterministic finite automata with counters from these expressions. The construction

[1] Measurements done with procps version 3.2.7 running GNU grep version 2.5.3 compiled with GNU cc version 4.1.2 on a machine with four 2,0 GHz 32-bit CPU running CentOS-5.2 with Linux 2.6.18 and GNU C library version 2.5.

can also be used to test in polynomial time whether a regular expression with numerical constraints is counter-1-unambiguous. The algorithm has been implemented[2] in `C` in a manner inspired by grep. The command in Fig. 1 executed with our implementation on the same machine uses less memory by three orders of magnitude.

The next section describes the regular expressions with numerical constraints, the languages they denote, and the 1-unambiguous regular expressions. Section 3 describes the finite automata with counters and shows an example of such an automaton. Section 4 shows how to construct a finite automaton with counters from a regular expression, and defines the counter-1-unambiguous regular expressions. The article ends with a section on related work and a conclusion.

2 Regular Expressions with Numerical Constraints

Fix an alphabet Σ and let $\mathbb{N} = \{1, 2, \ldots\}$ be the positive integers and $\mathbb{N}_{/1} = \{2, 3, 4, ...\} \cup \{\infty\}$.

Definition 1. *[6,7] Given an alphabet Σ, $\mathcal{R}_\Sigma$ is the set of (non-empty)* regular expressions with numerical constraints *over Σ, defined in the following manner:*

$$\mathcal{R}_\Sigma ::= \mathcal{R}_\Sigma + \mathcal{R}_\Sigma \,|\, \mathcal{R}_\Sigma \cdot \mathcal{R}_\Sigma \,|\, \mathcal{R}_\Sigma^{\mathbb{N}..\mathbb{N}_{/1}} \,|\, \Sigma \,|\, \epsilon$$

We disallow expressions of the form $r^{n..m}$ where $n > m$. We will use the abbreviations r^n for $r^{n..n}$, $r^{0..u}$ for $\epsilon + r^{1..u}$, $r^{n..}$ for $r^{n..\infty}$, r^+ for $r^{1..}$, and r^* for $r^{0..}$. Intuitively, $r^{n..}$ means that subexpression r must be matched n or more times, while $r^{n..m}$ means that r must be matched at least n and at most m times. In this paper, "regular expression" will mean regular expressions with numerical constraints.

The set of symbols from the alphabet occurring in a regular expression r, is denoted $\mathsf{sym}(r)$. We lift concatenation of words to sets of words, such that if $L_1, L_2 \subseteq \Sigma^*$, then $L_1 \cdot L_2 = \{w_1 \cdot w_2 \,|\, w_1 \in L_1 \wedge w_2 \in L_2\}$. Moreover, ϵ denotes the *empty word* of zero length, such that for all $w \in \Sigma^*$, $\epsilon \cdot w = w \cdot \epsilon = w$. Further, we allow non-negative integers as exponents meaning repeated concatenation, such that for any $L \subseteq \Sigma^*$, we have $L^n = L^{n-1} \cdot L$ for $n > 0$ and $L^0 = \{\epsilon\}$. For convenience, we recall in Definition 2 the language denoted by a regular expression, and extend it to numerical constraints. Since we will compare arbitrary members of $\mathbb{N}$ and $\mathbb{N}_{/1}$ below, we define that $i < \infty$ for all $i \in \mathbb{N}$.

Definition 2 (Language). *The language $L(r)$ denoted by a regular expression $r \in \mathcal{R}_\Sigma$, is defined in the following inductive way:*

$$\begin{aligned}
&L(r_1 + r_2) = L(r_1) \cup L(r_2)\\
&L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)\\
&L(r^{n..m}) = \textstyle\bigcup_{n \le i \le m} L(r)^i\\
&\text{for } a \in \Sigma \cup \{\epsilon\},\ L(a) = \{a\}
\end{aligned}$$

Some examples of regular expressions and their languages are: $L((a+b)^{0..2}) = \{\epsilon, a, b, aa, ab, ba, bb\}$ and $L((a^2b)^2) = \{aabaab\}$.

[2] Available from `http://www.ii.uib.no/~dagh/fac`

2.1 Term Trees and Positions

Given a regular expression r, we follow Terese [8] and define the term tree of r as the tree where the root is labelled with the main operator (choice, concatenation or numerical constraint) and the subtrees are the term trees of the subexpression(s) combined by the operator. If $a \in \Sigma \cup \{\epsilon\}$ the term tree is a single root-node with a as label.

We use $\langle n_1, \ldots, n_k \rangle$, a possibly empty sequence of natural numbers, to denote a position in a term tree. We let p, q, including subscripted variants, be variables for such possibly empty sequences of natural numbers. The position of the root is $\langle\rangle$. If $r = r_1 \cdot r_2$ or $r = r_1 + r_2$, and $n_1 \in \{1, 2\}$, the position $\langle n_1, \ldots, n_k \rangle$ in r is the position $\langle n_2, \ldots, n_k \rangle$ in the subtree of child n_1, that is, in the term tree of r_{n_1}. If $r = {r_1}^{l..u}$, the position $\langle 1, n_2, \ldots, n_k \rangle$ in r is the position $\langle n_2, \ldots, n_k \rangle$ in r_1, and $\langle 2 \rangle$ and $\langle 3 \rangle$ are the positions of the nodes containing the lower and upper limits l and u, respectively. For two positions $p = \langle m_1, \ldots, m_k \rangle$ and $q = \langle n_1, \ldots, n_l \rangle$, the notation $p \odot q$ will be used for the concatenated position $\langle m_1, \ldots, m_k, n_1, \ldots, n_l \rangle$. For a position p in r we will denote the subexpression rooted at this position by $r|_p$. Note that $r|_{\langle\rangle} = r$. Let $\mathsf{pos}(r)$ be the set of positions in r.

Note that for $r \in \mathcal{R}_\Sigma$, $p \in \mathsf{pos}(r)$, and $q \in \mathsf{pos}(r|_p)$, we have $r|_{p \odot q} = r|_p|_q$. This can be shown by induction on $r|_p$ (see, e.g., Terese [8]).

The concept of *marked expressions* will be important in this article. It has been used by Kilpeläinen & Tuhkanen [7] and by Brüggemann-Klein & Wood [9], but the definition given here is somewhat different.

Definition 3 (Marked Expressions). *If $r \in \mathcal{R}_\Sigma$ is a regular expression, $\mu(r) \in \mathcal{R}_{\mathsf{pos}(r)}$ is the marked expression, that is, the expression where every instance of any symbol from Σ is substituted with its position in the expression.*

It follows that if $p \in \mathsf{sym}(\mu(r))$, then $r|_p \in \mathsf{sym}(r)$. Note that, e.g., $\mu(b) = \mu(a) = \langle\rangle$, which shows that marking is not injective.

Example 1. As an example, consider $\Sigma = \{a, b, c\}$ and $r = (a^2 + bc)^{3..5}$. Then $\mu(r) = (\langle 1,1,1 \rangle^2 + \langle 1,2,1 \rangle \cdot \langle 1,2,2 \rangle)^{3..5}$. The term trees of r and $\mu(r)$ are shown in Fig. 2.

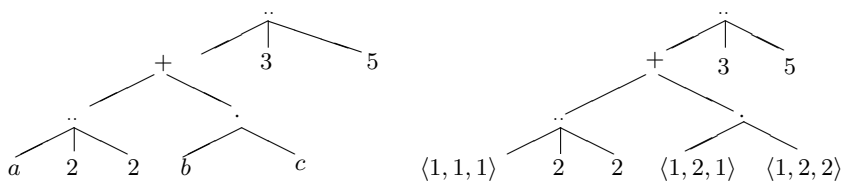

Fig. 2. Term trees for $(a^2 + bc)^{3..5}$ and $\mu((a^2 + bc)^{3..5})$

2.2 1-unambiguous Regular Expressions

Definition 4. *[5,9] A regular expression r is 1-unambiguous if for any two $upv, uqw \in L(\mu(r))$, where $p, q \in \mathsf{sym}(\mu(r))$ and $u, v, w \in \mathsf{sym}(\mu(r))^*$ such that $r|_p = r|_q$, we have $p = q$.*

Examples of 1-unambiguous regular expressions are $(a^{1..2})^{1..2}$ and $b^*a(b^*a)^*$, while $(\epsilon + a)a$ and $(a + b)^*a$ are not 1-unambiguous. The languages denoted by 1-unambiguous regular expressions without numerical constraints will be called *1-unambiguous regular languages*. Brüggemann-Klein & Wood [9] showed that there exist regular languages that are not 1-unambiguous regular languages, e.g. $L((a + b)^*(ac + bd))$. However, it is easy to modify a searching algorithm to search backwards, and the reverse of $(a+b)^*(ac+bd)$, namely $(ca+db)(a+b)^*$ is 1-unambiguous. There are of course also expressions like $(a+b)^*(ac+bd)(c+d)^*$, which denotes a 1-ambiguous language, read both backwards and forwards.

3 Finite Automata with Counters

3.1 Counter States and Update Instructions

We define *counter states*, which will be used to keep track of the number of times subexpressions with numerical constraints have been matched. Let $\mathcal{C}$ be the set of positions of subexpressions we need to keep track of. Let the mapping $\gamma : \mathcal{C} \mapsto \mathbb{N}$ denote a counter state. Let γ_1 be the counter state that maps all members of the domain to 1. We define an *update instruction* ψ as a partial mapping from $\mathcal{C}$ to $\{\mathsf{inc}, \mathsf{res}\}$ (inc for *increment*, res for *reset*). Update instructions ψ define mappings f_ψ between counter states in the following way: If $\psi(p) = \mathsf{inc}$, then $f_\psi(\gamma)(p) = \gamma(p) + 1$, if $\psi(p) = \mathsf{res}$ then $f_\psi(\gamma)(p) = 1$, and otherwise $f_\psi(\gamma)(p) = \gamma(p)$. Furthermore, we define the *counter-conditions* min and max, which map each member of $\mathcal{C}$ to lower and upper limits, respectively, such that $\mathsf{min}(p) \leq \mathsf{max}(p)$ for all $p \in \mathcal{C}$.

Definition 5 (Satisfaction of Update Instructions). *We define a satisfaction relation between update instructions, counter states and the two counter-conditions. Given $\mathsf{min} : \mathcal{C} \mapsto \mathbb{N}$, $\mathsf{max} : \mathcal{C} \mapsto \mathbb{N}_{/1}$, $\gamma : \mathcal{C} \mapsto \mathbb{N}$ and $\psi : \mathcal{C} \mapsto \{\mathsf{inc}, \mathsf{res}\}$, then $(\gamma, \mathsf{min}, \mathsf{max}) \models \psi$ holds if and only if the following holds for all p in the domain of ψ: whenever $\psi(p) = \mathsf{inc}$, then $\gamma(p) < \mathsf{max}(p)$, and whenever $\psi(p) = \mathsf{res}$, then $\gamma(p) \geq \mathsf{min}(p)$.*

The intuition of Definition 5 is that the value of a counter state can only be increased if the value is smaller than the maximum allowed value, while a value can only be reset if it is at least as large as the minimum value.

Example 2. Assume $C = \{p_1, p_2\}$, $\mathsf{min}(p_1) = \mathsf{max}(p_1) = 2$, $\mathsf{min}(p_2) = 1$, $\mathsf{max}(p_2) = \infty$ and $\gamma = \{p_1 \mapsto 2, p_2 \mapsto 1\}$, and let $\psi_1 = \{p_1 \mapsto \mathsf{inc}\}, \psi_2 = \{p_1 \mapsto \mathsf{res}, p_2 \mapsto \mathsf{inc}\}$ and $\psi_3 = \{p_1 \mapsto \mathsf{res}, p_2 \mapsto \mathsf{res}\}$. Then $f_{\psi_1}(\gamma) = \{p_1 \mapsto 3, p_2 \mapsto 1\}$, $f_{\psi_2}(\gamma) = \{p_1 \mapsto 1, p_2 \mapsto 2\}$ and $f_{\psi_3}(\gamma) = \{p_1 \mapsto 1, p_2 \mapsto 1\}$. Furthermore, $(\gamma, \mathsf{min}, \mathsf{max}) \models \psi_2$ and $(\gamma, \mathsf{min}, \mathsf{max}) \models \psi_3$ hold, while it does not hold that $(\gamma, \mathsf{min}, \mathsf{max}) \models \psi_1$.

3.2 Overlapping Update Instructions

Given mappings max and min, two update instructions are called *overlapping*, if there is a counter state that satisfies both of the update instructions.

Definition 6 (Overlapping Update Instructions). *Given mappings* max *and* min*, update instructions* ψ_1 *and* ψ_2 *are overlapping, if and only if there is a counter state* γ*, such that both* $(\gamma, \mathsf{min}, \mathsf{max}) \models \psi_1$ *and* $(\gamma, \mathsf{min}, \mathsf{max}) \models \psi_2$ *hold.*

Whether two update instructions are overlapping can be decided in linear time, relative to the size of $\mathcal{C}$, by the algorithm presented in the following proposition.

Proposition 1. *Given mappings* max *and* min*, two update instructions are overlapping if and only if: for every* p *that is mapped to different values by the two update instructions, it must hold that* $\mathsf{min}(p) < \mathsf{max}(p)$*.*

Proof. The proof is by treating the two parts of "if and only if" separately. First assume that for every p which is mapped to different values by the two update instructions, it holds that $\mathsf{min}(p) < \mathsf{max}(p)$. We must show that the update instructions are overlapping. A counter state γ satisfying both update instructions can be constructed as follows: For each member p of $\mathcal{C}$, if p is mapped to res by at least one of the update instructions, then let $\gamma(p) = \mathsf{min}(p)$, otherwise let $\gamma(p) = 1$. For the second part, that is, the "only if"-part of the proposition, assume the update instructions are overlapping. Thus there is at least one counter state γ which satisfies both update instructions ψ_1 and ψ_2. Now, for every p such that $\psi_1(p) = \mathsf{inc}$ and $\psi_2(p) = \mathsf{res}$, we get that $\mathsf{min}(p) \leq \gamma(p) < \mathsf{max}(p)$ from Definition 5, such that $\mathsf{min}(p) < \mathsf{max}(p)$. The opposite case where $\psi_1(p) = \mathsf{res}$ and $\psi_2(p) = \mathsf{inc}$ follows by symmetry. □

Recall Example 2. ψ_1 and ψ_2 are not overlapping, while ψ_3 is overlapping with ψ_2. The counter state satisfying both ψ_2 and ψ_3 constructed as in the argument above is γ. ψ_1 and ψ_3 are not overlapping.

3.3 Finite Automata with Counters

Definition 7 (Finite Automata with Counters). *A* Finite Automaton with Counters *(FAC) is a tuple* $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$*. The members of the tuple are summarized in Table 1 and described below:*

- Σ *is a finite, non-empty set (the alphabet).*
- Q *and* $\mathcal{C}$ *are finite sets of* states *and* counters*, respectively.*
- $q^I \in Q$ *is the* initial state*.*
- $\mathcal{A} : Q \mapsto \Sigma$ *maps each non-initial state to the letter which is matched when entering the state.*
- Φ *maps each state to a set of pairs. The latter pairs consist of a state and an update instruction.*

$$\Phi : Q \mapsto \wp(Q \times (\mathcal{C} \mapsto \{\mathsf{res}, \mathsf{inc}\}))\,.$$

Table 1. The members of the tuple describing an FAC

Symbol	Short description	Formally
Σ	Alphabet	Finite set
Q	States	Finite set
$\mathcal{C}$	Counters	Finite set
$\mathcal{A}$	Matching letter	$Q \mapsto \Sigma$
Φ	Transitions	$Q \mapsto \wp(Q \times (\mathcal{C} \mapsto \{\mathsf{res}, \mathsf{inc}\}))$
min	Counter minimum	$\mathcal{C} \mapsto \mathbb{N}$
max	Counter maximum	$\mathcal{C} \mapsto \mathbb{N}_{/1}$
q^I	Initial state	$q^I \in Q$
$\mathcal{F}$	Final configurations	$Q \mapsto \wp(\mathcal{C}) \cup \{\bot\}$

- $\mathsf{min} : \mathcal{C} \mapsto \mathbb{N}$ *and* $\mathsf{max} : \mathcal{C} \mapsto \mathbb{N}_{/1}$ *are the counter-conditions.*
- $\mathcal{F} : Q \mapsto \wp(\mathcal{C}) \cup \{\bot\}$ *describes the* final configurations *(See Definition 8). The symbol* $\bot$ *is used to indicate that a configuration is not final.*

Running or executing an FAC is defined in terms of *transitions* between *configurations*. The configurations of an FAC are pairs, where the first element is a member of Q, and the second element is a counter state.

Definition 8 (Configuration of an FAC). *A* configuration *of an FAC is a pair* (q, γ)*, where* $q \in Q$ *is the current state and* $\gamma : \mathcal{C} \mapsto \mathbb{N}$ *is the counter state. A configuration* (q, γ) *is final, if* $\mathcal{F}(q) \neq \bot$*, and for all* $c \in \mathcal{F}(q)$*,* $(\gamma, \mathsf{min}, \mathsf{max}) \models \{c \mapsto \mathsf{res}\}$*. Thus,* $\mathcal{F}(q)$ *specifies which counters should be "resettable".*

Intuitively, the first member of each of the pairs mapped to by Φ, is the state that can be entered, and the second member describes the changes to the current counter state of the automaton in this step. Thus, Φ and $\mathcal{A}$ together describe the possible transitions of the automaton. This is formalized as the transition function δ.

Definition 9 (Transition Function of an FAC). *For an FAC* $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$*, the transition function* δ *is defined for any configuration* (q, γ) *and letter* l *by*

$$\delta((q, \gamma), l) = \{(p, f_\psi(\gamma)) \mid \mathcal{A}(p) = l \wedge (p, \psi) \in \Phi(q) \wedge (\gamma, \mathsf{min}, \mathsf{max}) \models \psi\}.$$

Definition 10 (Deterministic FAC). *An FAC* $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$ *is deterministic if and only if* $|\delta((q, \gamma), l)| \leq 1$ *for all* $q \in Q, l \in \Sigma$ *and* $\gamma : \mathcal{C} \mapsto \mathbb{N}$*.*

Deciding whether an FAC is deterministic can be done in polynomial time as follows: For each state p, for each two different $(p_1, \psi_1), (p_2, \psi_2)$ both in $\Phi(p)$, assure that either $\mathcal{A}(p_1) \neq \mathcal{A}(p_2)$ or, otherwise, that ψ_1 and ψ_2 are not overlapping. That this test is sound and complete follows by the definition of δ and the properties of overlapping update instructions.

3.4 Word Recognition

Given a word as input, an FAC can either *accept* or *reject* this. A deterministic FAC recognizes a word by treating letters in the word one by one. It starts in the *initial configuration* (q^I, γ_1). An FAC in configuration (q, γ), with letter $l \in \Sigma$ next in the word, will reject the word if $\delta((q, \gamma), l)$ is empty. Otherwise it enters the unique configuration $(q', \gamma') \in \delta((q, \gamma), l)$. If the whole word has been read, a deterministic FAC accepts the word if and only if it is in a final configuration. The subset of Σ^* consisting of words being accepted by an FAC A is denoted $L(A)$.

Example 3. Let $\Sigma = \{a, b, c\}$, $Q = \{q^I, \langle 1,1,1\rangle, \langle 1,2,1\rangle, \langle 1,2,2\rangle\}$ and $\mathcal{C} = \{\langle\rangle, \langle 1,1\rangle\}$. Figure 3 illustrates a deterministic FAC $(\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$ which recognizes $L((a^2 + bc)^{3..5})$. The sequence of configurations of this FAC while recognizing *aabcaa* is :

$$
\begin{array}{l}
(q^I, \gamma_1) \\
(\langle 1,1,1\rangle, \{\langle\rangle \mapsto 1, \langle 1,1\rangle \mapsto 1\}) \\
(\langle 1,1,1\rangle, \{\langle\rangle \mapsto 1, \langle 1,1\rangle \mapsto 2\}) \\
(\langle 1,2,1\rangle, \{\langle\rangle \mapsto 2, \langle 1,1\rangle \mapsto 1\}) \\
(\langle 1,2,2\rangle, \{\langle\rangle \mapsto 2, \langle 1,1\rangle \mapsto 1\}) \\
(\langle 1,1,1\rangle, \{\langle\rangle \mapsto 3, \langle 1,1\rangle \mapsto 1\}) \\
(\langle 1,1,1\rangle, \{\langle\rangle \mapsto 3, \langle 1,1\rangle \mapsto 2\})
\end{array}
$$

The last configuration is final, since $\mathsf{min}(\langle\rangle) \leq 3$ and $\mathsf{min}(\langle 1,1\rangle) \leq 2$.

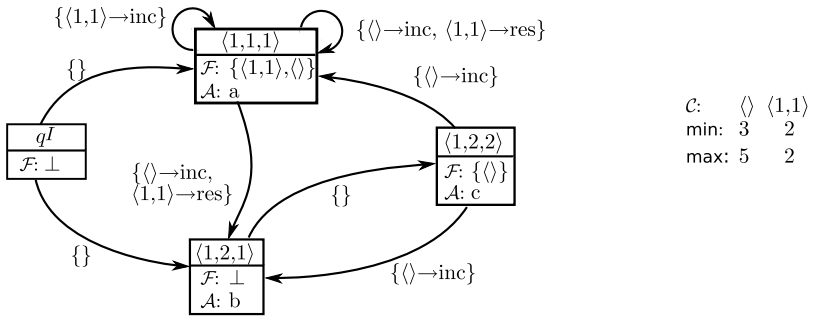

Fig. 3. Illustration of FAC recognizing $L((a^2 + bc)^{3..5})$. Every state is depicted as a rectangle separated in two by a line. The name of the state is in the upper part of the rectangle, and the values of $\mathcal{F}$ and $\mathcal{A}$ are in the lower part. Every member of ϕ is shown as an arrow, annotated with the corresponding update instruction. $\mathcal{C}$, min and max are shown on the right hand side.

Proposition 2 (Linear-Time Recognition). *For any textual representation of FACs, and for any deterministic FAC* $A = (\Sigma, Q, \mathcal{C}, \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$*, if* $\sigma(A)$ *is the size of the textual representation of* A*, then for any word* $w \in \Sigma^*$*, the FAC* A *can in time* $O(|w|\sigma(A)^2)$ *decide whether* w *is rejected or accepted.*

Proof. The FAC makes no more than $|w|$ steps in the recognition, and at each step, there can be no more than $\max\{|\Phi(q)| \mid q \in Q\}$ outgoing edges, and for each of these we might have to check the counter state γ against no more than $|\mathcal{C}|$ constraints. Testing whether the last configuration is accepting, takes time $O(|\mathcal{C}| \cdot \max\{|\mathcal{F}(q)| \mid q \in Q\})$. Thus we get the result, as $|\mathcal{C}|$, $\max\{|\mathcal{F}(q)| \mid q \in Q\}$ and $\max\{|\Phi(q)| \mid q \in Q\}$ are all $O(\sigma(A))$.

3.5 Searching with FACs

We formalize the problems called matching and searching as the binary predicates $\mathsf{m}, \mathsf{s} \subseteq \mathcal{R}_\Sigma \times \Sigma^*$, defined as follows: $\mathsf{m}(r, w) \Leftrightarrow w \in L(r)$ and $\mathsf{s}(r, w) \Leftrightarrow \exists u, v, v' : (w = u \cdot v \cdot v' \wedge v \in L(r))$. A deterministic FAC recognizing $L(r)$ can decide $\mathsf{m}(r, w)$ in time linear in $|w|$. If the alphabet (Σ) is fixed, we can solve $\mathsf{s}(r, w)$ in time linear in the length of w by solving $\mathsf{m}(\Sigma^* \cdot r \cdot \Sigma^*)$, where Σ here denotes the disjunction of all the letters. In practical cases, though, the size of Σ can be prohibitively large. Another option is therefore to decide $\mathsf{s}(r, w)$ by using $O(|w|^2)$ executions of an algorithm for m. A deterministic FAC can also decide in linear time the *prefix problem*. The latter is also formalized as a binary predicate, namely $\mathsf{p} \subseteq \mathcal{R}_\Sigma \times \Sigma^*$, where $\mathsf{p}(r, w) \Leftrightarrow \exists u, v : (w = u \cdot v \wedge u \in L(r))$. $O(|w|)$ executions of an algorithm for p is sufficient to decide s. Thus, deterministic FACs can be used to search in time quadratic in the length of the text. The last approach is similar to that used in GNU grep.

4 Constructing Finite Automata with Counters

Following Brüggemann-Klein & Wood [9] and Glushkov [10], we define three mappings, first, last, and follow. They will be used below in an alternative definition of the language denoted by a regular expression, and will be central in the construction of FACs from regular expressions. first takes a regular expression as parameter and returns the set of positions that could be matching the first letter in a word in the language of the regular expression. Similarly, the mapping last takes a regular expression as parameter and returns the set of positions that could be matching the last letter in a word in the language of the regular expression.

follow takes a regular expression and a position in the expression as parameters, and returns a set of pairs (p, ψ). Assume the position given as argument to follow is used to match a letter in a word in the language of the regular expression. If follow returns a set containing (p, ψ), then p is a position in the regular expression which could match the next letter in the word, and ψ is the update instructions, describing what changes must be done to the counters in the step to p from the position given as the second argument.

Before we can define first, last and follow, we need some auxiliary definitions.

Definition 11 (Concatenating Positions with Update Instructions and Sets of Positions).

- *For* $p \in \mathbb{N}^*$ *and* $S \subseteq \mathbb{N}^*$, *let* $p \odot S = \{p \odot q \mid q \in S\}$
- *For* $p \in \mathbb{N}^*$ *and* $\psi : (\mathbb{N}^* \mapsto \{\mathsf{res}, \mathsf{inc}\})$, *let* $p \odot \psi = \{p \odot q \mapsto \psi(q) \mid q \in \mathsf{dom}(\psi)\}$.
- *For* $S \subseteq \mathbb{N}^* \times (\mathbb{N}^* \mapsto \{\mathsf{inc}, \mathsf{res}\})$ *let* $p \odot S = \{(p \odot q, p \odot \psi) \mid (q, \psi) \in S\}$.

Definition 12 (Subposition). *We use the notation* $p \leq q$ *for* p *a* prefix *or* subposition *of* q, *that is,* $p \leq q \Leftrightarrow \exists p_1 : q = p \odot p_1$.

Definition 13. *Let* $r \in \mathcal{R}_\Sigma$ *and* $q \in \mathsf{pos}(r)$.

1. *Let* $\mathsf{C}(r) \subseteq \mathsf{pos}(r)$ *be the positions of all subexpressions of* r *that are of the form* $r_1^{n..m}$, *and that are not of the form* r_1^+. *Expressed formally,*

$$\mathsf{C}(r) = \{q \in \mathsf{pos}(r) \mid \exists n \in \mathbb{N}, m \in \mathbb{N}_{/1}, r_1 \in \mathcal{R}_\Sigma : r|_q = r_1^{n..m} \neq r_1^+\}.$$

2. *Let* $\mathsf{C}(r, q) \subseteq \mathsf{C}(r)$ *be the set of positions in* $\mathsf{C}(r)$ *above* q, *that is,* $\mathsf{C}(r, q) = \{p \in \mathsf{C}(r) \mid p \leq q\}$.

In the sequel we need to express the set of regular expressions whose language contains the empty word. The set of nullable expressions, $\mathfrak{N}_\Sigma$, is therefore defined as follows:

Definition 14 (Nullable Expressions). *Given an alphabet* Σ, *the set of nullable expressions,* $\mathfrak{N}_\Sigma$, *is defined in the following inductive manner*

$$\mathfrak{N}_\Sigma ::= \mathfrak{N}_\Sigma \cdot \mathfrak{N}_\Sigma \mid \mathfrak{N}_\Sigma + \mathcal{R}_\Sigma \mid \mathcal{R}_\Sigma + \mathfrak{N}_\Sigma \mid \mathfrak{N}_\Sigma^{\mathbb{N}..\mathbb{N}_{/1}} \mid \epsilon$$

We can prove that $\mathfrak{N}_\Sigma = \{r \in \mathcal{R}_\Sigma \mid \epsilon \in L(r)\}$ by induction on r.

We will define inductively $\mathsf{first} : \mathcal{R}_\Sigma \mapsto \wp(\mathbb{N}^*)$ (Table 2), $\mathsf{last} : \mathcal{R}_\Sigma \mapsto \wp(\mathbb{N}^*)$ (Table 2) and $\mathsf{follow} : (\mathcal{R}_\Sigma \times \mathbb{N}^*) \mapsto \wp(\mathbb{N}^* \times (\mathbb{N}^* \mapsto \{\mathsf{res}, \mathsf{inc}\}))$ (Table 3). first and last map from an expression r to a subset of $\mathsf{sym}(\mu(r))$, such that $\mathsf{first}(r) = \{p \in \mathsf{sym}(\mu(r)) \mid \exists w \in \mathsf{sym}(\mu(r))^* : pw \in L(\mu(r))\}$ and $\mathsf{last}(r) = \{p \in \mathsf{sym}(\mu(r)) \mid \exists w \in \mathsf{sym}(\mu(r))^* : wp \in L(\mu(r))\}$. follow maps an expression r and a position $q \in \mathsf{pos}(r)$ to a set of pairs of the form (p, ψ), where $p \in \mathsf{sym}(\mu(r))$ and $\psi : \mathsf{C}(r) \mapsto \{\mathsf{inc}, \mathsf{res}\}$.

Recall the example expression $r = (a^2 + bc)^{3..5}$ from Example 1. We get $\mathsf{first}(r) = \{\langle 1, 1, 1\rangle, \langle 1, 2, 1\rangle\}$, $\mathsf{last}(r) = \{\langle 1, 1, 1\rangle, \langle 1, 2, 2\rangle\}$, and follow is shown in Table 4.

4.1 Basic Properties

The following lemma basically summarizes that first, last and follow have the intended properties.

Lemma 1. *For all regular expressions* $r \in \mathcal{R}_\Sigma$ *and all positions* $q \in \mathsf{sym}(\mu(r))$:

Table 2. $\mathsf{first} : \mathcal{R}_\Sigma \mapsto \wp(\mathbb{N}^*)$ and $\mathsf{last} : \mathcal{R}_\Sigma \mapsto \wp(\mathbb{N}^*)$

$\mathsf{first}(\epsilon) = \mathsf{last}(\epsilon) = \varnothing, \qquad r \in \Sigma \Rightarrow \mathsf{first}(r) = \mathsf{last}(r) = \{\langle\rangle\}$
$r = r_1 + r_2 \Rightarrow$ $\mathsf{first}(r) = (\langle 1\rangle \odot \mathsf{first}(r_1)) \cup (\langle 2\rangle \odot \mathsf{first}(r_2))$ $\wedge\ \mathsf{last}(r) = (\langle 1\rangle \odot \mathsf{last}(r_1)) \cup (\langle 2\rangle \odot \mathsf{last}(r_2))$
$r = r_1 \cdot r_2 \wedge r_1 \in \mathfrak{N}_\Sigma \ \Rightarrow\ \mathsf{first}(r) = (\langle 1\rangle \odot \mathsf{first}(r_1)) \cup (\langle 2\rangle \odot \mathsf{first}(r_2))$ $r = r_1 \cdot r_2 \wedge r_2 \in \mathfrak{N}_\Sigma \ \Rightarrow\ \mathsf{last}(r) = (\langle 1\rangle \odot \mathsf{last}(r_1)) \cup (\langle 2\rangle \odot \mathsf{last}(r_2))$
$r = r_1 \cdot r_2 \wedge r_1 \notin \mathfrak{N}_\Sigma \ \Rightarrow\ \mathsf{first}(r) = \langle 1\rangle \odot \mathsf{first}(r_1)$ $r = r_1 \cdot r_2 \wedge r_2 \notin \mathfrak{N}_\Sigma \ \Rightarrow\ \mathsf{last}(r) = \langle 2\rangle \odot \mathsf{last}(r_2)$
$r = r_1^{n..m} \ \Rightarrow\ \mathsf{first}(r) = \langle 1\rangle \odot \mathsf{first}(r_1) \ \wedge\ \mathsf{last}(r) = \langle 1\rangle \odot \mathsf{last}(r_1)$

Table 3. $\mathsf{follow} : (\mathcal{R}_\Sigma \times \mathbb{N}^*) \mapsto \wp(\mathbb{N}^* \times (\mathbb{N}^* \mapsto \{\mathsf{res}, \mathsf{inc}\}))$

$r \in \Sigma \ \Rightarrow\ \mathsf{follow}(r, \langle\rangle) = \varnothing$
$r = r_1 + r_2 \ \Rightarrow\ \big(\mathsf{follow}(r, \langle 1\rangle \odot q) = \langle 1\rangle \odot \mathsf{follow}(r_1, q)\big)$ $\wedge\ \big(\mathsf{follow}(r, \langle 2\rangle \odot q) = \langle 2\rangle \odot \mathsf{follow}(r_2, q)\big)$
$r = r_1 \cdot r_2 \ \Rightarrow\ \mathsf{follow}(r, \langle 2\rangle \odot q) = \langle 2\rangle \odot \mathsf{follow}(r_2, q)$
$r = r_1 \cdot r_2 \wedge\ q \in \mathsf{last}(r_1) \ \Rightarrow$ $\mathsf{follow}(r, \langle 1\rangle \odot q) = \langle 1\rangle \odot \mathsf{follow}(r_1, q) \cup$ $\{(q_1, \{\langle 1\rangle \odot p_1 \mapsto \mathsf{res} \,|\, p_1 \in \mathsf{C}(r_1, q)\}) \,|\, q_1 \in \langle 2\rangle \odot \mathsf{first}(r_2)\}$
$r = r_1 \cdot r_2 \wedge\ q \notin \mathsf{last}(r_1) \ \Rightarrow\ \mathsf{follow}(r, \langle 1\rangle \odot q) = \langle 1\rangle \odot \mathsf{follow}(r_1, q)$
$r = r_1^+ \wedge\ q \in \mathsf{last}(r_1) \ \Rightarrow\ \mathsf{follow}(r, \langle 1\rangle \odot q) =$ $\langle 1\rangle \odot \mathsf{follow}(r_1, q) \cup \big\{\ \big(q_1,\ \{\langle 1\rangle \odot p_1 \mapsto \mathsf{res} \,|\, p_1 \in \mathsf{C}(r_1, q)\}\big) \,\big|\, q_1 \in \langle 1\rangle \odot \mathsf{first}(r_1)\ \big\}$
$r = r_1^{n..m} \wedge\ q \in \mathsf{last}(r_1) \wedge\ (n, m) \neq (1, \infty) \ \Rightarrow$ $\mathsf{follow}(r, \langle 1\rangle \odot q) = \langle 1\rangle \odot \mathsf{follow}(r_1, q) \cup$ $\big\{\ \big(q_1,\ \{\langle\rangle \mapsto \mathsf{inc}\} \cup \{\langle 1\rangle \odot p_1 \mapsto \mathsf{res} \,|\, p_1 \in \mathsf{C}(r_1, q)\}\big) \,\big|\, q_1 \in \langle 1\rangle \odot \mathsf{first}(r_1)\ \big\}$
$r = r_1^{n..m} \wedge\ q \notin \mathsf{last}(r_1) \ \Rightarrow\ \mathsf{follow}(r, \langle 1\rangle \odot q) = \langle 1\rangle \odot \mathsf{follow}(r_1, q)$

Table 4. The mapping follow for $r = (a^2 + bc)^{3..5}$

$$\mathsf{follow}(r, \langle 1,1,1\rangle) = \left\{ \begin{array}{l} (\langle 1,1,1\rangle, \{\langle 1,1\rangle \mapsto \mathsf{inc}\}), \\ (\langle 1,1,1\rangle, \{\langle 1,1\rangle \mapsto \mathsf{res}, \langle\rangle \mapsto \mathsf{inc}\}), \\ (\langle 1,2,1\rangle, \{\langle 1,1\rangle \mapsto \mathsf{res}, \langle\rangle \mapsto \mathsf{inc}\}) \end{array} \right\}$$

$$\mathsf{follow}(r, \langle 1,2,1\rangle) = \{(\langle 1,2,2\rangle, \{\})\}$$

$$\mathsf{follow}(r, \langle 1,2,2\rangle) = \{(\langle 1,1,1\rangle, \{\langle\rangle \mapsto \mathsf{inc}\}), (\langle 1,2,1\rangle, \{\langle\rangle \mapsto \mathsf{inc}\})\}$$

1. $\mathsf{first}(r) = \{p \in \mathsf{sym}(\mu(r)) \mid \exists w \in \mathsf{sym}(\mu(r))^* : pw \in L(\mu(r))\}$
2. $\mathsf{last}(r) = \{p \in \mathsf{sym}(\mu(r)) \mid \exists w \in \mathsf{sym}(\mu(r))^* : wp \in L(\mu(r))\}$
3. $\mathsf{follow}(r, q)$ *is well-defined.*
4. $\forall (p, \psi) \in \mathsf{follow}(r, q) : \exists u, v \in \mathsf{sym}(\mu(r))^* : uqpv \in L(\mu(r))$
5. $\forall (p, \psi) \in \mathsf{follow}(r, q) : \forall q' \in \mathsf{C}(r) :$

$$\begin{aligned} & q' \notin \mathsf{dom}(\psi) \Rightarrow (q' \notin \mathsf{C}(r, q) \vee (\exists u, v \in \mathsf{sym}(\mu(r)|_{q'})^* : uqpv \in L(\mu(r)|_{q'}))) \\ \wedge \quad & \psi(q') = \mathsf{inc} \Rightarrow (q \in q' \odot \mathsf{last}(r|_{q'}) \wedge p \in q' \odot \mathsf{first}(r|_{q'})) \\ \wedge \quad & \psi(q') = \mathsf{res} \Leftrightarrow (q \in q' \odot \mathsf{last}(r|_{q'}) \wedge q' \notin \mathsf{C}(r, p)) \end{aligned}$$

All items can be proved by induction on r, using the preceding items. The proofs are omitted for space considerations.

Theorem 1 (Polynomial Runtime). *For all regular expressions $r \in \mathcal{R}_\Sigma$ and all positions $q \in \mathsf{sym}(\mu(r))$:*

1. *Computing* $\mathsf{first}(r)$ *and* $\mathsf{last}(r)$ *takes time* $O(|r|)$.
2. *Computing* $\mathsf{follow}(r, q)$ *for all* q*, takes time* $O(|r|^3)$.

Proof. 1. For part 1 note first that $|\mathsf{first}(r)| = O(|r|)$ and $|\mathsf{last}(r)| = O(|r|)$ follows from parts 1 and 2 of Lemma 1. We will assume that union of sets can be done in linear time, and that prefixing a number to a position (as in $\langle 1 \rangle \odot p$) can be done in constant time. We can then show that the run-time is $O(|r|)$ by induction on r.

2. For part 2, start with computing first and last for all subexpressions of r. This takes time $O(|r|^3)$. Computing $\mathsf{follow}(r, q)$ will then mean a linear number of *calls* to follow, each of which takes maximally $O(|r|^2)$ time in addition to the recursive call to follow. □

4.2 Counter-1-unambiguity

We can now define the right unambiguity we need for constructing deterministic automata. *Counter-1-unambiguous* regular expressions are introduced in this section. Section 4.3 describes how a deterministic FAC can be constructed in polynomial time from such expressions. However, the construction of FACs can be applied to regular expressions in a larger class, namely, the regular expressions in *constraint normal form*. The construction of an FAC from an expression in this class can also be done in polynomial time, but the FAC might not be deterministic. An expression is in constraint normal form if, for every subexpression of the form $r^{n..m}$, r is not nullable.

Definition 15. *A regular expression r is in constraint normal form if and only if there is no subexpression of r of the form $r_1^{n..m}$ where $r_1 \in \mathfrak{N}_\Sigma$.*

For example, $(a^*a)^{2..3}$ is in constraint normal form, while $(a^*)^{2..3}$ is not.

Given a regular expression r, let mappings $\mathsf{min} : \mathsf{C}(r) \mapsto \mathbb{N}$ and $\mathsf{max} : \mathsf{C}(r) \mapsto \mathbb{N}_{/1}$ be such that $\mathsf{min}(q) = r|_{q \odot \langle 2 \rangle}$ and $\mathsf{max}(q) = r|_{q \odot \langle 3 \rangle}$, and define a binary relation $\simeq$ between the pairs $\mathsf{sym}(\mu(r)) \times (\mathcal{C} \mapsto \{\mathsf{inc}, \mathsf{res}\})$, where $(q_2, \psi_2) \simeq (q_1, \psi_1)$ if and only if $r|_{q_2} = r|_{q_1}$ and ψ_1 and ψ_2 are overlapping update instructions (as according to Definition 6).

Definition 16 (Counter-1-unambiguity). *A regular expression r in constraint normal form is* counter-1-unambiguous*, if* $\forall p, q \in \mathsf{first}(r) : r|_p = r|_q \Rightarrow p = q$ *and* $\forall q \in \mathsf{sym}(\mu(r)) : \forall (q_2, \psi_2), (q_1, \psi_1) \in \mathsf{follow}(r, q) : (q_2, \psi_2) \simeq (q_1, \psi_1) \Rightarrow (q_2, \psi_2) = (q_1, \psi_1)$.

The regular expressions used as examples in Sect. 1 are counter-1-unambiguous. Examples of expressions that are not counter-1-unambiguous are $(a^{1..2})^{1..2}$, $(a^*a)^{2..3}$ and $(a^{1..2} + b)^{1..2}$, while $(a + b)^{1..4}$ is counter-1-unambiguous. For some of the expressions that are not counter-1-unambiguous, we can multiply the numerical constraints to possibly get counter-1-unambiguous expressions. In general, for regular expressions of the form $(r^{l_1..u_1})^{l_2..u_2}$, if $l_2 \geq \frac{l_1 - 1}{u_1 - l_1}$, then $L(r^{l_1 \cdot l_2 .. u_1 \cdot u_2}) = L((r^{l_1..u_1})^{l_2..u_2})$. For example, $L((a^{1..2})^{1..2}) = L(a^{1..4})$.

4.3 Constructing FACs

Given a regular expression r and the mappings first, last and follow as defined above, we construct the $\mathsf{FAC}(r)$, an FAC $(\Sigma, Q, \mathsf{C}(r), \mathcal{A}, \Phi, \mathsf{min}, \mathsf{max}, q^I, \mathcal{F})$, where $Q = \mathsf{sym}(\mu(r)) \cup \{q^I\}$ and where min and max are as above. $\forall q \in \mathsf{sym}(\mu(r))$, let $\mathcal{A}(q) = r|_q$ and $\Phi(q) = \mathsf{follow}(r, q)$. Let $\Phi(q^I) = \{(q, \varnothing) \mid q \in \mathsf{first}(r)\}$.

The initial configuration is final if and only if r is nullable. For the other configurations, two conditions must be met: the position the current state represents must be in $\mathsf{last}(r)$, and the numerical constraints containing this position must be satisfied. Thus, the mapping $\mathcal{F}$ is defined as follows. Let first $\mathcal{F}' = \{p \mapsto \mathsf{C}(r, p) \mid p \in \mathsf{last}(r)\} \cup \{q \mapsto \bot \mid q \in \mathsf{sym}(\mu(r)) - \mathsf{last}(r)\}$. If $r \in \mathfrak{N}_\Sigma$, then let $\mathcal{F} = \mathcal{F}' \cup \{q^I \mapsto \varnothing\}$, and otherwise let $\mathcal{F} = \mathcal{F}' \cup \{q^I \mapsto \bot\}$.

Figure 4 illustrates some properties of this algorithm. The result of applying this algorithm to $r = (a^2 + bc)^{3..5}$ from Example 1 is the FAC in Example 3.

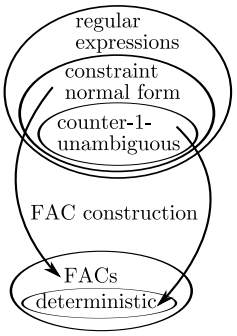

Fig. 4. Some properties of the construction of FACs.

4.4 Equivalence of $L(r)$ and $L'(r)$

We will now define $L'(r)$, which is the language recognized by the FAC constructed from r as described above.

Definition 17 ($L'(r)$). *For $r \in \mathcal{R}_\Sigma$, $L'(r)$ is the subset of Σ^*, such that $\epsilon \in L'(r)$ iff $r \in \mathfrak{N}_\Sigma$ and for all $w \in L'(r)$ where $n = |w| > 0$, there exist $p_1, \ldots, p_n \in \mathsf{sym}(\mu(r))$, and if $n > 1$ there are also $\psi_2, \ldots, \psi_n \in (\mathsf{C}(r) \mapsto \{\mathsf{inc}, \mathsf{res}\})$, such that all of the following five items hold:*

1. $r|_{p_1} \cdots r|_{p_n} = w$.
2. $p_1 \in \mathsf{first}(r)$.
3. $p_n \in \mathsf{last}(r)$.

4. *If* $n > 1$, *then* $\forall i \in \{1, \ldots, n-1\} : (p_{i+1}, \psi_{i+1}) \in \mathsf{follow}(r, p_i)$.
5. $\forall i \in \{1, \ldots, n\} : (f_{\psi_i}(\cdots f_{\psi_1}(\gamma_1)\cdots), \mathsf{min}, \mathsf{max}) \models \psi_{i+1}$, *where* $\psi_1 = \varnothing$, $\psi_{n+1} = \{p \mapsto \mathsf{res} \mid p \in \mathsf{C}(r, p_n)\}$.

Theorem 2. *If* $r \in \mathcal{R}_\Sigma$ *is in constraint normal form, then* $L(r) = L'(r)$.

The proof is by induction on r and uses the definitions of $L'(r)$ and $L(r)$ and the facts in Lemma 1. The proof is omitted for space considerations.

5 Related Work and Conclusion

5.1 Related Work

The inspiration for Finite Automata with Counters comes, of course, from finite automata as defined, e.g., by Hopcroft & Ullman [11], by Kleene [12] or by Glushkov [10]. Kilpeläinen & Tuhkanen [4,7,13], and Gelade et al. [6] also investigated properties of the regular expressions with counters, and give algorithms for membership, and definitions of automata classes for regular expressions with numerical constraints. Tuhkanen & Kilpeläinen's counter automata seem to handle a larger class of expressions than FACs, but they are not defined formally, only by examples. The technical report referred to in the paper was never finished (personal communication). Tuhkanen & Kilpeläinen's counter automata also differ from FACs in the way iterations are kept track of, with extra states, called "levels".

Colazzo, Ghelli & Sartiani describe in [14] an algorithm for linear-time membership in a subclass of regular expressions called collision-free. The collision-free regular expressions have at most one occurrence of each symbol from Σ, and the counters (and the Kleene star) can only be applied directly to letters or disjunctions of letters. The latter class is smaller than, and included in, the class of counter-1-unambiguous regular expressions. The main focus of Colazzo, Ghelli & Sartiani is on the extension of regular expressions used in *XML Schemas*. This extension includes *interleaving*, which is not covered by the algorithm presented here.

A class of tree automata with counting are described by Zilio & Lugiez [15]. Our approach also has similarities to the tagged automata found in Laurikari [16]. The results by Brüggemann-Klein & Wood in [5,9,17] concerning 1-unambiguous regular expressions, are in some ways what the current article attempts to extend to the regular expressions with counters.

5.2 Conclusion

We have defined *Finite Automata with Counters* (FAC), and a translation from the regular expressions with numerical constraints to these automata. We defined *constraint normal form*, a subset of the regular expressions with numerical constraints, for which the translation to FACs can be done in polynomial time. Further we defined *counter-1-unambiguous regular expressions*, a subset of the

regular expressions of constraint normal form, and for which the FAC resulting from the translation is deterministic. The deterministic FAC can recognize the language of the given regular expression in time linear in the size of word to be tested. Testing whether an FAC is deterministic can be done in polynomial time.

References

1. The Open Group: The Open Group Base Specifications Issue 6, IEEE Std 1003.1. 2 edn. (1997)
2. GNU: GNU grep manual.
3. Fallside, D.C.: XML Schema part 0: Primer, W3C recommendation. Technical report, World Wide Web Consortium (W3C) (2001)
4. Kilpeläinen, P., Tuhkanen, R.: Regular expressions with numerical occurrence indicators - preliminary results. In Kilpeläinen, P., Päivinen, N., eds.: SPLST, University of Kuopio, Department of Computer Science (2003) 163–173
5. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoretical Computer Science **120**(2) (1993) 197–213
6. Gelade, W., Martens, W., Neven, F.: Optimizing schema languages for XML: Numerical constraints and interleaving. In Schwentick, T., Suciu, D., eds.: Proceedings of ICDT. Volume 4353 of Lecture Notes in Computer Science., Springer (2007) 269–283
7. Kilpeläinen, P., Tuhkanen, R.: One-unambiguity of regular expressions with numeric occurrence indicators. Information and Computation **205**(6) (2007) 890–916
8. Bezem, M., Klop, J.W., de Vrijer, R., eds.: Term Rewriting Systems. Cambridge University Press (2003)
9. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Information and Computation **140**(2) (1998) 229–253
10. Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys **16**(5) (1961) 1–53
11. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
12. Kleene, S.C.: Representation of events in nerve sets and finite automata. Automata Studies (1956) 3–41
13. Kilpeläinen, P., Tuhkanen, R.: Towards efficient implementation of XML schema content models. In Munson, E.V., Vion-Dury, J.Y., eds.: ACM Symposium on Document Engineering, ACM (2004) 239–241
14. Ghelli, G., Colazzo, D., Sartiani, C.: Linear time membership in a class of regular expressions with interleaving and counting. In Shanahan, J.G., Amer-Yahia, S., Manolescu, I., Zhang, Y., Evans, D.A., Kolcz, A., Choi, K.S., Chowdhury, A., eds.: CIKM, ACM (2008) 389–398
15. Dal-Zilio, S., Lugiez, D.: Xml schema, tree logic and sheaves automata. In Nieuwenhuis, R., ed.: RTA. Volume 2706 of Lecture Notes in Computer Science., Springer (2003) 246–263
16. Laurikari, V.: NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In: SPIRE. (2000) 181–187
17. Brüggemann-Klein, A.: Regular expressions into finite automata. In Simon, I., ed.: LATIN. Volume 583 of Lecture Notes in Computer Science., Springer (1992) 87–98