Cuts in Regular Expressions

Martin Berglund[1], Henrik Björklund[1], Frank Drewes[1],
Brink van der Merwe[2], and Bruce Watson[2]

[1] Umeå University, Sweden
{mbe,henrikb,drewes}@cs.umu.se
[2] Stellenbosch University, South Africa
abvdm@cs.sun.ac.za, bruce@fastar.org

Abstract. Most software packages with regular expression matching engines offer operators that extend the classical regular expressions, such as counting, intersection, complementation, and interleaving. Some of the most popular engines, for example those of Java and Perl, also provide operators that are intended to control the nondeterminism inherent in regular expressions. We formalize this notion in the form of the *cut* and *iterated cut* operators. They do not extend the class of languages that can be defined beyond the regular, but they allow for exponentially more succinct representation of some languages. Membership testing remains polynomial, but emptiness testing becomes PSPACE-hard.

1 Introduction

Regular languages are not only a theoretically well-understood class of formal languages. They also appear very frequently in real world programming. In particular, regular expressions are a popular tool for solving text processing problems. For this, the ordinary semantics of regular expressions, according to which an expression simply denotes a language, is extended by an informally defined operational understanding of how a regular expression is "applied" to a string. The usual default in regular expression matching libraries is to search for the leftmost matching substring, and pick the longest such substring [2]. This behavior is often used to repeatedly match different regular expressions against a string (or file contents) using program control flow to decide the next expression to match. Consider the repeatedly matching pseudo-code below, and assume that `match_regex` matches the longest prefix possible:

```
match = match_regex("(a*b)*", s);
if(match != null) then
    if(match_regex("ab*c", match.string_remainder) != null) then
        return match.string_remainder == "";
return false;
```

For the string $s = abac$, this program first matches $R_1 = (a^* \cdot b)^*$ to the substring ab, leaving ac as a remainder, which is matched by $R_2 = a \cdot (b^*) \cdot c$, returning true.

M.-P. Béal and O. Carton (Eds.): DLT 2013, LNCS 7907, pp. 70–81, 2013.

The set of strings s for which the program returns "true" is in fact a regular language, but it is *not* the regular language defined by $R_1 \cdot R_2$. Consider for example the string $s = aababcc$, which is matched by $R_1 \cdot R_2$. However, in an execution of the program above, R_1 will match $aabab$, leaving the remainder cc, which is not matched by R_2. The expression $R_1 \cdot R_2$ exhibits non-deterministic behavior which is lost in the case of the earliest-longest-match strategy combined with the explicit *if*-statement. This raises the question, are programs of this type (with arbitrarily many *if*-statements freely nested) always regular, and how can we describe the languages they recognize?

Related Work. Several extensions of regular expressions that are frequently available in software packages, such as counting (or numerical occurrence indicators, not to be confused with counter automata), interleaving, intersection, and complementation, have been investigated from a theoretical point of view. The succinctness of regular expressions that use one or more of these extra operators compared to standard regular expressions and finite automata were investigated, e.g., in [4, 6, 8]. For regular expressions with intersection, the membership problem was studied in, e.g., [10, 14], while the equivalence and emptiness problems were analyzed in [3, 15]. Interleaving was treated in [5, 11] and counting in [9, 12]. To our knowledge, there is no previous theoretical treatment of the cut operator introduced in this paper, or of other versions of possessive quantification.

Paper Outline. In the next section we formalize the control of nondeterminism outlined above by defining the cut and iterated cut operators, which can be included directly into regular expressions, yielding so-called cut expressions. In Section 3, we show that adding the new operators does not change the expressive power of regular expressions, but that it does offer improved succinctness. Section 4 provides a polynomial time algorithm for the uniform membership problem of cut expressions, while Section 5 shows that emptiness is PSPACE-hard. In Section 6, we compare the cut operator to the similar operators found more or less commonly in software packages in the wild (Perl, Java, PCRE, etc.). Finally, Section 7 summarizes some open problems.

2 Cut Expressions

We denote the natural numbers (including zero) by $\mathbb{N}$. The set of all strings over an alphabet Σ is denoted by Σ^*. In particular, Σ^* contains the empty string ε. The set $\Sigma^* \setminus \{\varepsilon\}$ is denoted by Σ^+. We write $\mathit{pref}(u)$ to denote the set of nonempty prefixes of a string u and $\mathit{pref}_\varepsilon(u)$ to denote $\mathit{pref}(u) \cup \{\varepsilon\}$. The canonical extensions of a function $f\colon A \to B$ to a function from A^* to B^* and to a function from 2^A to 2^B are denoted by f as well.

As usual, a regular expression over an alphabet Σ (where $\varepsilon, \emptyset \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \mid E')$, $(E \cdot E')$, or (E^*). Parentheses can be dropped using the rule that * (Kleene closure[1])

[1] Recall that the Kleene closure of a language L is the smallest language L^* such that $\{\varepsilon\} \cup LL^* \subseteq L^*$.

takes precedence over $\cdot$ (concatenation), which takes precedence over $|$ (union). Moreover, outermost parentheses can be dropped, and $E \cdot E'$ can be written as EE'. The language $\mathcal{L}(E)$ denoted by a regular expression is obtained by evaluating E as usual, where $\emptyset$ stands for the empty language and $a \in \Sigma \cup \{\varepsilon\}$ for $\{a\}$. We denote by $E \equiv E'$ the fact that two regular expressions (or, later on, cut expressions) E and E' are equivalent, i.e., that $\mathcal{L}(E) = \mathcal{L}(E')$. Where the meaning is clear from context we may omit the $\mathcal{L}$ and write E to mean $\mathcal{L}(E)$.

Let us briefly recall finite automata. A nondeterministic finite automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ consisting of a finite set Q of states, a initial state $q_0 \in Q$, a set $F \subseteq Q$ of final states, an alphabet Σ, and a transition function $\delta\colon Q \times \Sigma \to 2^Q$. In the usual way, δ extends to a function $\delta\colon \Sigma^* \to 2^Q$, i.e., $\delta(\varepsilon) = \{q_0\}$ and $\delta(wa) = \bigcup_{q \in \delta(w)} \delta(q, a)$. A accepts $w \in \Sigma^*$ if and only if $\delta(w) \cap F \neq \emptyset$, and it recognizes the language $\mathcal{L}(A) = \{w \in \Sigma^* \mid \delta(w) \cap F \neq \emptyset\}$. A deterministic finite automaton (DFA) is the special case where $|\delta(q, a)| \leq 1$ for all $(q, a) \in Q \times \Sigma$. In this case we consider δ to be a function $\delta\colon Q \times \Sigma \to Q$, so that its canonical extension to strings becomes a function $\delta\colon Q \times \Sigma^* \to Q$.

We now introduce cuts, iterated cuts, and cut expressions. Intuitively, $E\,!\,E'$ is the variant of EE' in which E greedily matches as much of a string as it can accommodate, leaving the rest to be matched by E'. The so-called iterated cut $E^{!*}$ first lets E match as much of a string as possible, and seeks to iterate this until the whole string is matched (if possible).

Definition 1 (cut and cut expression). *The* cut *is the binary operation* $!$ *on languages such that, for languages* L, L'*,*

$$L\,!\,L' = \{uv \mid u \in L, v \in L', uv' \notin L \text{ for all } v' \in \mathit{pref}(v)\}.$$

The iterated cut of L*, denoted by* $L^{!*}$*, is the smallest language that satisfies*

$$\{\varepsilon\} \cup (L\,!\,(L^{!*})) \subseteq L^{!*}$$

(i.e., $L\,!\,(L\,!\cdots(L\,!\,(L\,!\,\{\varepsilon\}))\cdots) \subseteq L^{!*}$ *for any number of repetitions of the cut).*

Cut expressions *are expressions built using the operators allowed in regular expressions, the cut, and the iterated cut. A cut expression denotes the language obtained by evaluating that expression in the usual manner.*

The precedence rules give $^{!*}$ precedence over $\cdot$, which in turn gets precedence over $!$ which in turn gets precedence over $|$.

The motivation for the inclusion of the iterated cut is two-fold; (i) it is a natural extension for completeness in that it relates to the cut like the Kleene closure relates to concatenation; and, (ii) in the context of a program like that shown on page 1, the iterated cut permits the modelling of matching regular expressions in loops.

Let us discuss a few examples.

1. The cut expression $ab^*\,!\,b$ yields the empty language. This is because every string in $\mathcal{L}(ab^*b)$ is in $\mathcal{L}(ab^*)$ as well, meaning that the greedy matching of

the first subexpression will never leave a b over for the second. Looking at the definition of the cut, a string in $\mathcal{L}(ab^* \,!\, b)$ would have to be of the form ub, such that $u \in \mathcal{L}(ab^*)$ but $ub \notin \mathcal{L}(ab^*)$. Clearly, such a string does not exist. More generally, if $\varepsilon \notin \mathcal{L}(E')$ then $\mathcal{L}(E \,!\, E') \subseteq \mathcal{L}(EE') \setminus \mathcal{L}(E)$. However, as the next example shows, the converse inclusion does not hold.

2. We have $(a^* \mid b^*) \,!\, (ac \mid bc) \equiv a^+bc \mid b^+ac$.[2] This illustrates that the semantics of the cut cannot be expressed by concatenating subsets of the involved languages. In the example, there are no subsets L_1 and L_2 of $\mathcal{L}(a^* \mid b^*)$ and $\mathcal{L}(ac \mid bc)$, respectively, such that $L_1 \cdot L_2 = \mathcal{L}(a^* \mid b^*) \,!\, \mathcal{L}(ac \mid bc)$.
3. Clearly, $((ab)^* \,!\, a) \,!\, b \equiv (ab)^*ab$ whereas $(ab)^* \,!\, (a \,!\, b) \equiv (ab)^* \,!\, ab \equiv \emptyset$ (as in the first example). Thus, the cut is not associative.
4. As an example of an iterated cut, consider $((aa)^* \,!\, a)^*$. We have $(aa)^* \,!\, a \equiv (aa)^*a$ and therefore $((aa)^* \,!\, a)^* \equiv a^*$. This illustrates that matching a string against $(E \,!\, E')^*$ *cannot* be done by greedily matching E, then matching E', and iterating this procedure. Instead, one has to "chop" the string to be matched into substrings and match each of those against $E \,!\, E'$. In particular, $(E \,!\, \varepsilon)^* \equiv E^*$ (since $E \,!\, \varepsilon \equiv E$). This shows that $E^{!*}$ cannot easily be expressed by means of cut and Kleene closure.
5. Let us finally consider the interaction between the Kleene closure and the iterated cut. We have $L^{!*} \subseteq L^*$ and thus $(L^{!*})^* \subseteq (L^*)^* = L^*$. Conversely, $L \subseteq L^{!*}$ yields $L^* \subseteq (L^{!*})^*$. Thus $(L^{!*})^* = L^*$ for all languages L.
   Similarly, we also have $(L^*)^{!*} = L^*$. Indeed, if $w \in L^*$, then it belongs to $(L^*)^{!*}$, since the first iteration of the iterated cut can consume all of w. Conversely, $(L^*)^{!*} \subseteq (L^*)^* = L^*$. Thus, altogether $(L^*)^{!*} = L^* = (L^{!*})^*$

3 Cut Expressions versus Finite Automata

In this section, we compare cut expressions and finite automata. First, we show that the languages described by cut expressions are indeed regular. We do this by showing how to convert cut expressions into equivalent finite automata. Second, we show that cut expressions are succinct: There are cut expressions containing only a single cut (and no iterated cut), such that a minimal equivalent NFA or regular expression is of exponential size.

3.1 Cut Expressions Denote Regular Languages

Let A, A' be DFAs. To prove that the languages denoted by cut expressions are regular, it suffices to show how to construct DFAs recognizing $L(A) \,!\, L(A')$ and $L(A)^{!*}$. We note here that an alternative proof would be obtained by showing how to construct alternating automata (AFAs) recognizing $L(A) \,!\, L(A')$ and $L(A)^{!*}$. Such a construction would be slightly simpler, especially for the iterated cut, but since the conversion of AFAs to DFAs causes a doubly exponential size increase [1], we prefer the construction given below, which (almost) saves one

[2] As usual, we abbreviate EE^* by E^+.

level of exponentiality. Moreover, we hope that this construction, though more complex, is more instructive.

We first handle the comparatively simple case $L(A)\,!\,L(A')$. The idea of the construction is to combine A with a kind of product automaton of A and A'. The automaton starts working like A. At the point where A reaches one of its final states, A' starts running in parallel with A. However, in contrast to the ordinary product automaton, the computation of A' is reset to its initial state whenever A reaches one of its final states again. Finally, the string is accepted if and only if A' is in one of its final states.

To make the construction precise, let $A = (Q, \Sigma, \delta, q_0, F)$ and $A' = (Q', \Sigma, \delta', q_0', F')$. In order to disregard a special case, let us assume that $q_0 \notin F$. (The case where $q_0 \in F$ is easier, because it allows us to use only product states in the automaton constructed.) We define a DFA $\overline{A} = (\overline{Q}, \Sigma, \overline{\delta}, q_0, \overline{F})$ as follows:

- $\overline{Q} = Q \cup (Q \times Q')$ and $\overline{F} = Q \times F'$,
- for all $q, r \in Q$, $\overline{q} = (q, q') \in \overline{Q}$, and $a \in \Sigma$ with $\delta(q, a) = r$

$$\overline{\delta}(q,a) = \begin{cases} r & \text{if } r \notin F \\ (r, q_0') & \text{otherwise,} \end{cases} \quad \text{and} \quad \overline{\delta}(\overline{q},a) = \begin{cases} (r, \delta'(q',a)) & \text{if } r \notin F \\ (r, q_0') & \text{otherwise.} \end{cases}$$

Let $w \in \Sigma^*$. By construction, $\overline{\delta}$ has the following properties:

1. If $u \notin L(A)$ for all $u \in \mathit{pref}_\varepsilon(w)$, then $\overline{\delta}(w) = \delta(w)$.
2. Otherwise, let $w = uv$, where u is the longest prefix of w such that $u \in L(A)$. Then $\overline{\delta}(w) = (\delta(w), \delta'(v))$.

We omit the easy inductive proof of these statements. By the definition of $L(A)\,!\,L(A')$ and the choice of $\overline{F}$, they imply that $L(\overline{A}) = L(A)\,!\,L(A')$. In other words, we have the following lemma.

Lemma 2. *For all regular languages L and L', the language $L\,!\,L'$ is regular.*

Let us now consider the iterated cut. Intuitively, the construction of a DFA recognizing $L(A)^{!*}$ is based on the same idea as above, except that the product construction is iterated. The difficulty is that the straightforward execution of this construction yields an infinite automaton. For the purpose of exposing the idea, let us disregard this difficulty for the moment. Without loss of generality, we assume that $q_0 \notin F$ (which we can do because $L(A)^{!*} = (L(A) \setminus \{\varepsilon\})^{!*}$) and that $\delta(q, a) \neq q_0$ for all $q \in Q$ and $a \in \Sigma$.

We construct an automaton whose states are strings $q_1 \cdots q_k \in Q^+$. The automaton starts in state q_0, initially behaving like A. If it reaches one of the final states of A, say q_1, it continues in state $q_1 q_0$, working essentially like the automaton for $L(A)\,!\,L(A)$. In particular, it "resets" the second copy each time the first copy encounters a final state of A. However, should the second copy reach a final state q_2 of A (while $q_1 \notin F$), a third copy is spawned, thus resulting in a state of the form $q_1 q_2 q_0$, and so on.

Formally, let $\delta_a \colon Q \to Q$ be given by $\delta_a(q) = \delta(q, a)$ for all $a \in \Sigma$ and $q \in Q$. Recall that functions to extend to sequences, so $\delta_a : Q^* \to Q^*$ operates elementwise. We construct the (infinite) automaton $\widehat{A} = (\widehat{Q}, \Sigma, \widehat{\delta}, q_0, \widehat{F})$ as follows:

– $\widehat{Q} = (Q \setminus \{q_0\})^* Q$.
– For all $s = q_1 \cdots q_k \in \widehat{Q}$ and $a \in \Sigma$ with $\delta_a(s) = q'_1 \cdots q'_k$

$$\widehat{\delta}(s,a) = \begin{cases} q'_1 \cdots q'_k & \text{if } q'_1, \ldots, q'_k \notin F \\ q'_1 \cdots q'_l q_0 & \text{if } l = \min\{i \in \{1,\ldots,k\} \mid q'_i \in F\}. \end{cases} \quad (1)$$

– $\widehat{F} = \{q_1 \cdots q_k \in \widehat{Q} \mid q_k = q_0\}$.

Note that $\widehat{\delta}(s,a) \in \widehat{Q}$ since we assume that $\delta(q,a) \neq q_0$ for all $q \in Q$ and $a \in \Sigma$.

Similar to the properties of $\overline{A}$ above, we have the following:

Claim 1. Let $w = v_1 \cdots v_k \in \Sigma^*$, where $v_1 \cdots v_k$ is the unique decomposition of w such that (a) for all $i \in \{1,\ldots,k-1\}$, v_i is the longest prefix of $v_i \cdots v_k$ which is in $L(A)$ and (b) $\mathit{pref}_\varepsilon(v_k) \cap L(A) = \emptyset$.[3] Then $\widehat{\delta}(w) = \delta(v_1 \cdots v_k)\delta(v_2 \cdots v_k) \cdots \delta(v_k)$. In particular, $\widehat{A}$ accepts w if and only if $w \in L(A)^{!*}$.

Again, we omit the straightforward inductive proof.

It remains to be shown how to turn the set of states of $\widehat{A}$ into a finite set. We do this by verifying that repetitions of states of A can be deleted. To be precise, let $\pi(s)$ be defined as follows for all $s = q_1 \cdots q_k \in \widehat{Q}$. If $k = 1$ then $\pi(s) = s$. If $k > 1$ then

$$\pi(s) = \begin{cases} \pi(q_1 \cdots q_{k-1}) & \text{if } q_k \in \{q_1, \ldots, q_{k-1}\} \\ \pi(q_1 \cdots q_{k-1}) q_k & \text{otherwise.} \end{cases}$$

Let $\pi(\widehat{A})$ be the NFA obtained from $\widehat{A}$ by taking the quotient with respect to π, i.e., by identifying all states $s, s' \in \widehat{Q}$ such that $\pi(s) = \pi(s')$. The set of final states of $\pi(\widehat{A})$ is the set $\pi(\widehat{F})$.

This completes the construction. The following lemmas prove its correctness.

Lemma 3. *For all $s \in \widehat{Q}$ and $a \in \Sigma$ it holds that $\pi(\widehat{\delta}(s,a)) = \pi(\widehat{\delta}(\pi(s),a))$.*

Proof. By the very definition of π, for every function $f\colon Q \to Q$ and all $s \in \widehat{Q}$ we have $\pi(f(s)) = \pi(f(\pi(s)))$. In particular, this holds for $f = \delta_a$. Now, let $s = q_1 \cdots q_k$ be as in the definition of $\widehat{\delta}$. Since the same set of symbols occurs in $\delta_a(s)$ and $\delta_a(\pi(s))$, the same case of Equation 1 applies for the construction of $\widehat{\delta}(s,a)$ and $\widehat{\delta}(\pi(s),a)$. In the first case $\pi(\widehat{\delta}(s,a)) = \pi(\delta_a(s)) = \pi(\delta_a(\pi(s))) = \pi(\widehat{\delta}(\pi(s),a))$. In the second case

$$\begin{aligned} \pi(\widehat{\delta}(s,a)) &= \pi(\delta_a(q_1 \cdots q_l) q_0) \\ &= \pi(\delta_a(q_1 \cdots q_l)) q_0 \\ &= \pi(\delta_a(\pi(q_1 \cdots q_l))) q_0 \\ &= \pi(\delta_a(\pi(q_1 \cdots q_l)) q_0) \\ &= \pi(\widehat{\delta}(\pi(s),a)). \end{aligned}$$

Note that the second and the fourth equality make use of the fact that $q_0 \notin \{q_1, \ldots, q_{k-1}\}$, which prevents π from deleting the trailing q_0. □

[3] The strings $v_1, \ldots, v_k$ are well defined because $\varepsilon \notin L(A)$.

Lemma 4. *The automaton $\pi(\widehat{A})$ is a DFA such that $L(\pi(\widehat{A})) = L(A)^{!*}$. In particular, $L^{!*}$ is regular for all regular languages L.*

Proof. To see that $\pi(\widehat{A})$ is a DFA, let $a \in \Sigma$. By the definition of $\pi(\widehat{A})$, its transition function $\widehat{\delta}_\pi$ is given by

$$\widehat{\delta}_\pi(t, a) = \{\pi(\widehat{\delta}(s, a)) \mid s \in \widehat{Q},\ t = \pi(s)\}$$

for all $t \in \pi(\widehat{Q})$. However, by Lemma 3, $\pi(\widehat{\delta}(s, a)) = \pi(\widehat{\delta}(t, a))$ is independent of the choice of s. In other words, $\widehat{A}$ is a DFA. Furthermore, by induction on the length of $w \in \Sigma^*$, Lemma 3 yields $\widehat{\delta}_\pi(w) = \pi(\widehat{\delta}(w))$. Thus, by Claim 1, $L(\pi(\widehat{A})) = L(A)^{!*}$. In particular, for a regular language L, this shows that $L^{!*}$ is regular, by picking A such that $\mathcal{L}(A) = L$. □

We note here that, despite the detour via an infinite automaton, the construction given above can effectively be implemented. Unfortunately, it results in a DFA of size $\mathcal{O}(n!)$, where n is the number of states of the original DFA.

Theorem 5. *For every cut expression E, $\mathcal{L}(E)$ is regular.*

Proof. Follows from combining Lemmas 2 and 4. □

3.2 Succinctness of Cut Expressions

In this section we show that for some languages, cut expressions provide an exponentially more compact representation than regular expressions and NFAs.

Theorem 6. *For every $k \in \mathbb{N}_+$, there exists a cut expression E_k of size $\mathcal{O}(k)$ such that every NFA and every regular expression for $\mathcal{L}(E_k)$ is of size $2^{\Omega(k)}$. Furthermore, E_k does not contain the iterated cut and it contains only one occurrence of the cut.*

Proof. We use the alphabets $\Sigma = \{0, 1\}$ and $\Gamma = \Sigma \cup \{], [\}$. For $k \in \mathbb{N}_+$, let

$$E_k = (\varepsilon \mid [\Sigma^* 0 \Sigma^{k-1} 1 \Sigma^*] \mid [\Sigma^* 1 \Sigma^{k-1} 0 \Sigma^*])\,![\Sigma^{2k}].$$

Each string in the language $\mathcal{L}(E_k)$ consists of one or two bitstrings enclosed in square brackets. If there are two, the first has at least two different bits at a distance of exactly k positions and the second is an arbitrary string in Σ^{2k}. However, when there is only a single pair of brackets the bitstring enclosed is of length $2k$ and its second half will be an exact copy of the first.

We argue that any NFA that recognizes $\mathcal{L}(E_k)$ must have at least 2^k states. Assume, towards a contradiction, that there is an NFA A with fewer than 2^k states that recognizes $\mathcal{L}(E_k)$.

Since $|\Sigma^k| = 2^k$ there must exist two distinct bitstrings w_1 and w_2 of length k such that the following holds. There exist a state q of A and accepting runs ρ_1 and ρ_2 of A on $[w_1 w_1]$ and $[w_2 w_2]$, resp., such that ρ_1 reaches q after reading $[w_1$

and ρ_2 reaches q after reading $[w_2$. This, in turn, means that there are accepting runs ρ_1' and ρ_2' of A_q on $w_1]$ and $w_2]$, respectively, where A_q is the automaton obtained from A by making q the sole initial state. Combining the first half of ρ_1 with ρ_2' gives an accepting run of A on $[w_1 w_2]$. This is a contradiction and we conclude that there is no NFA for E_k with fewer than 2^k states.

The above conclusion also implies that every regular expression for $\mathcal{L}(E_k)$ has size $2^{\Omega(k)}$. If there was a smaller regular expression, the Glushkov construction [7] would also yield a smaller NFA. □

Remark 7. The only current upper bound is the one implied by Section 3.1, from which automata of non-elementary size cannot be ruled out as it yields automata whose sizes are bounded by powers of twos.

A natural restriction on cut expressions is to only allow cuts to occur at the topmost level of the expression. This gives a tight bound on automata size.

Lemma 8. *Let E be a cut expression, without iterated cuts, such that no subexpression of the form C^* or $C \cdot C'$ contains cuts. Then the minimal equivalent DFA has $2^{\mathcal{O}(|E|)}$ states, and this bound is tight.*

Proof (sketch). Given any DFAs A, A', using product constructions we get DFAs for $L(A) \mid L(A')$ and $L(A) ! L(A')$ whose number of states is proportional to the product of the number of states in A and A'. (See Lemma 2 for the case $L(A) ! L(A')$.) Thus, one can construct an exponential-sized DFA in a bottom-up manner. Theorem 6 shows that this bound is tight. □

4 Uniform Membership Testing

We now present an easy membership test for cut expressions that uses a dynamic programming approach (or, equivalently, memoization). Similarly to the Cocke-Younger-Kasami algorithm, the idea is to check which substrings of the input string belong to the languages denoted by the subexpressions of the given cut expression. The pseudocode of the algorithm is shown in Algorithm 1. Here, the string $u = a_1 \cdots a_n$ to be matched against a cut expression E is a global variable. For $1 \leq i \leq j \leq n+1$, $\mathit{Match}(E, i, j)$ will check whether $a_i \cdots a_{j-1} \in \mathcal{L}(E)$. We assume that an implicit table is used in order to memoize computed values for a given input triple. Thus, recursive calls with argument triples that have been encountered before will immediately return the memoized value rather than executing the body of the algorithm.

Theorem 9. *The uniform membership problem for cut expressions can be decided in time $\mathcal{O}(m \cdot n^3)$, where m is the size of the cut expression and n is the length of the input string.*

Proof. Consider a cut expression E_0 of size m and a string $u = a_1 \cdots a_n$. It is straightforward to show by induction on $m+n$ that $\mathit{Match}(E, i, j) = \mathit{true}$ if and only if $a_i \cdots a_{j-1} \in \mathcal{L}(E)$, where $1 \leq i \leq j \leq n+1$ and E is a subexpression of

Algorithm 1. $Match(E, i, j)$

```
if E = ∅ then return false
else if E = ε then return i = j
else if E ∈ Σ then return j = i + 1 ∧ E = a_i
else if E = E_1 | E_2 then return Match(E_1, i, j) ∨ Match(E_2, i, j)
else if E = E_1 · E_2 then
    for k = 0, ..., j − i do
        if Match(E_1, i, i + k) ∧ Match(E_2, i + k, j) then return true
    return false
else if E = E_1^* then
    for k = 1, ..., j − i do
        if Match(E_1, i, i + k) ∧ Match(E, i + k, j) then return true
    return i = j
else if E = E_1 ! E_2 then
    for k = j − i, ..., 0 do
        if Match(E_1, i, i + k) then return Match(E_2, i + k, j)
    return false
else if E = E_1^{!*} then
    for k = j − i, ..., 1 do
        if Match(E_1, i, i + k) then return Match(E, i + k, j)
    return i = j
```

E_0. For $E = E_1 \,!\, E_2$, this is because of the fact that $v \in \mathcal{L}(E)$ if and only if v has a longest prefix $v_1 \in \mathcal{L}(E_1)$, and the corresponding suffix v_2 of v (i.e., such that $v = v_1 v_2$) is in $\mathcal{L}(E_2)$. Furthermore, it follows from this and the definition of the iterated cut that, for $E = E_1^{!*}$, $v \in \mathcal{L}(E)$ if either $v = \varepsilon$ or v has a longest prefix $v_1 \in \mathcal{L}(E_1)$ such that the corresponding suffix v_2 is in $\mathcal{L}(E)$.

Regarding the running time of $Match(E, 1, n+1)$, by memoization the body of *Match* is executed at most once for every subexpression of E and all i, j, $1 \leq i \leq j \leq n+1$. This yields $\mathcal{O}(m \cdot n^2)$ executions of the loop body. Moreover, a single execution of the loop body involves at most $\mathcal{O}(n)$ steps (counting each recursive call as one step), namely if $E = E_1^*$, $E = E_1 \,!\, E_2$ or $E = E^{!*}$. □

5 Emptiness Testing of Cut Expressions

Theorem 10. *Given a cut expression E, it is PSPACE-hard to decide whether $\mathcal{L}(E) = \emptyset$. This remains true if $E = E_1 \,!\, E_2$, where E_1 and E_2 are regular expressions.*

Proof. We prove the theorem by reduction from regular expression universality, i.e. deciding for a regular expression R and an alphabet Σ whether $\mathcal{L}(R) = \Sigma^*$. This problem is well known to be PSPACE-complete [12].Given R, we construct a cut expression E such that $\mathcal{L}(E) = \emptyset$ if and only if $\mathcal{L}(R) = \Sigma^*$.

We begin by testing if $\varepsilon \in \mathcal{L}(R)$. This can be done in polynomial time. If $\varepsilon \notin \mathcal{L}(R)$, then we set $E = \varepsilon$, satisfying $\mathcal{L}(E) \neq \emptyset$. Otherwise, we set $E = R \,!\, \Sigma$. If R is universal, there is no string ua such that $u \in \mathcal{L}(R)$ but $ua \notin \mathcal{L}(R)$. Thus $\mathcal{L}(E)$ is empty. If R is not universal, since $\varepsilon \in \mathcal{L}(R)$ there are $u \in \Sigma^*$ and $a \in \Sigma$ such that $u \in \mathcal{L}(R)$ and $ua \notin \mathcal{L}(R)$, which means that $ua \in \mathcal{L}(E) \neq \emptyset$. □

Lemma 11. *For cut expressions E the problems whether $\mathcal{L}(E) = \emptyset$ and $\mathcal{L}(E) = \Sigma^*$ are LOGSPACE-equivalent.*

Proof. Assume that $\# \notin \Sigma$, and let $\Sigma' = \Sigma \cup \{\#\}$. The lemma then follows from these two equivalences: (i) $E \equiv \emptyset$ if and only if $((\varepsilon \mid E\Sigma^*) \,!\, \Sigma^+) \mid \varepsilon \equiv \Sigma^*$; and; (ii) $E \equiv \Sigma^*$ if and only if $(\varepsilon \mid E\#(\Sigma')^*) \,!\, \Sigma^*\# \equiv \emptyset$. □

6 Related Concepts in Programming Languages

Modern regular expression matching engines have numerous highly useful features, some of which improve succinctness (short-hand operators) and some of which enable expressions that specify non-regular languages. Of interest here is that most regular expression engines in practical use feature at least *some* operation intended to control nondeterminism in a way that resembles the cut. They are however only loosely specified in terms of *backtracking*, the specific evaluation technique used by many regular expression engines. This, combined with the highly complex code involved, makes formal analysis difficult.

All these operations appear to trace their ancestry to the first edition of "Mastering Regular Expressions" [2], which contains the following statement:

> *"A feature I think would be useful, but that no regex flavor that I know of has, is what I would call possessive quantifiers. They would act like normal quantifiers except that once they made a decision that met with local success, they would never backtrack to try the other option. The text they match could be unmatched if their enclosing subexpression was unmatched, but they would never give up matched text of their own volition, even in deference to the overall match."[2]*

The cut operator certainly fits this somewhat imprecise description, but as we shall see implementations have favored different interpretations. Next we give a brief overview of three different operations implemented in several major regular expression engines, that exhibit some control over nondeterminism. All of these operators are of great practical value and are in use. Still, they feature some idiosyncrasies that should be investigated, in the interest of bringing proper regular behavior to as large a set of regular expression functionality as possible.

Possessive Quantifiers. Not long after the proposal for the possessive quantifier, implementations started showing up. It is available in software such as Java, PCRE, Perl, etc. For a regular expression R the operation is denoted R^{*+}, and behaves like R^* except it never backtracks. This is already troublesome, since "backtracking" is poorly defined at best, and, in fact, by itself $\mathcal{L}(R^{*+}) = \mathcal{L}(R^*)$, but $\mathcal{L}(R^{*+} \cdot R') = \mathcal{L}(R^* \,!\, R')$ for all R'. That is, extending regular expressions with possessive quantifiers makes it possible to write expressions such that $\mathcal{L}(E \cdot E') \neq \mathcal{L}(E) \cdot \mathcal{L}(E')$, an example being given by $E = a^{*+}$ and $E' = a$. This violates the compositional spirit of regular expressions.

Next, consider Table 1. The expression on the first row, call it R, is tested in each of the given implementations, and the language recognized is shown. The results on the first row are easy to accept from every perspective. The second

Table 1. Some examples of possessive quantifier use

Expression	Perl 5.16.2	Java 1.6.0u18	PCRE 8.32
$(aa)^{*+}a$	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$
$((aa)^{*+}a)^*$	$\{\varepsilon, a, aaa, aaaaa, \ldots\}$	$\{\varepsilon, a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$
$((aa)^{*+}a)^*a$	$\{a\}$	$\{a\}$	$\{a\}$

Table 2. Comparison between Perl and PCRE when using the (*PRUNE) operator

Expression	Perl 5.10.1	Perl 5.16.2	PCRE 8.32
$(aa)^*$(*PRUNE)a	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$
$((aa)^*$(*PRUNE)$a)^*$	$\{\varepsilon, a, aa, aaa, \ldots\}$	$\emptyset$	$\emptyset$
a^*(*PRUNE)a	$\{a, aa, aaa, \ldots\}$	$\emptyset$	$\emptyset$

row however has the expression R^*, and despite $a \in \mathcal{L}(R)$ no implementation gives $aa \in \mathcal{L}(R^*)$, which violates the classical compositional meaning of the Kleene closure (in addition, in PCRE we have $\varepsilon \notin \mathcal{L}(R^*)$). The third row further illustrates how the compositional view of regular expressions breaks down when using possessive quantifiers.

Independent Groups or Atomic Subgroups. A practical shortcoming of the possessive quantifiers is that the "cut"-like operation cannot be separated from the quantifier. For this reason most modern regular expression engines have also introduced atomic subgroups ("independent groups" in Java). An atomic subgroup containing the expression R is denoted $(?\rangle R)$, and described as "preventing backtracking". Any subexpression $(?\rangle R^*)$ is equivalent to R^{*+}, but subexpressions of the form $(?\rangle R)$ where the topmost operation in R is not a Kleene closure may be hard to translate into an equivalent expression using possessive quantifiers.

Due to the direct translation, atomic subgroups suffer from all the same idiosyncrasies as possessive quantifiers, such as $\mathcal{L}(((?\rangle(aa)^*)a)^*a) = \{a\}$.

*Commit Operators and (*PRUNE).* In Perl 6 several interesting "commit operators" relating to nondeterminism control were introduced. As Perl 5 remains popular they were back-ported to Perl 5 in version 5.10.0 with different syntax. The one closest to the pure cut is (*PRUNE), called a "zero-width pattern", an expression that matches ε (and therefore always succeeds) but has some engine side-effect. As with the previous operators the documentation depends on the internals of the implementation. *"[(*PRUNE)] prunes the backtracking tree at the current point when backtracked into on failure"*[13].

These operations are available both in Perl and PCRE, but interestingly their semantics in Perl 5.10 and Perl 5.16 differ in subtle ways; see Table 2. Looking at the first two rows we see that Perl 5.10 matches our compositional understanding of the Kleene closure (i.e., row two has the same behavior as $((aa)*!a)*$). On the other hand Perl 5.10 appears to give the wrong answer in the third row example.

7 Discussion

We have introduced cut operators and demonstrated several of their properties. Many open questions and details remain to be worked out however:

- There is a great distance between the upper and lower bounds on minimal automata size presented in Section 3.2, with an exponential lower bound for both DFA and NFA, and a non-elementary upper bound in general.
- The complexity of uniform membership testing can probably be improved as the approach followed by Algorithm 1 is very general. (It can do complementation, for example.)
- The precise semantics of the operators discussed in Section 6 should be studied further, to ensure that all interesting properties can be captured.

Acknowledgments. We thank Yves Orton who provided valuable information about the implementation and semantics of (*PRUNE) in Perl.

References

[1] Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM 28(1), 114–133 (1981)
[2] Friedl, J.E.F.: Mastering Regular Expressions. Reilly & Associates, Inc., Sebastopol (1997)
[3] Fürer, M.: The complexity of the inequivalence problem for regular expressions with intersection. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 234–245. Springer, Heidelberg (1980)
[4] Gelade, W.: Succinctness of regular expressions with interleaving, intersection and counting. Theor. Comput. Sci. 411(31-33), 2987–2998 (2011)
[5] Gelade, W., Martens, W., Neven, F.: Optimizing schema languages for XML: Numerical constraints and interleaving. SIAM J. Comput. 38(5), 2021–2043 (2009)
[6] Gelade, W., Neven, F.: Succinctness of the complement and intersection of regular expressions. ACM Trans. Comput. Logic 13(1), Article 4 (2012)
[7] Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys 16, 1–53 (1961)
[8] Gruber, H., Holzer, M.: Tight bounds on the descriptional complexity of regular expressions. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 276–287. Springer, Heidelberg (2009)
[9] Kilpeläinen, P., Tuhkanen, R.: Regular expressions with numerical occurrence indicators - preliminary results. In: SPLST, pp. 163–173 (2003)
[10] Kupferman, O., Zuhovitzky, S.: An improved algorithm for the membership problem for extended regular expressions. In: Diks, K., Rytter, W. (eds.) MFCS 2002. LNCS, vol. 2420, pp. 446–458. Springer, Heidelberg (2002)
[11] Mayer, A.J., Stockmeyer, L.J.: Word problems - this time with interleaving. Inform. and Comput. 115(2), 293–311 (1994)
[12] Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential time. In: SWAT (FOCS), pp. 125–129 (1972)
[13] Perl 5 Porters. perlre (2012), `http://perldoc.perl.org/perlre.html` (accessed January 16, 2013)
[14] Petersen, H.: The membership problem for regular expressions with intersection is complete in LOGCFL. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 513–522. Springer, Heidelberg (2002)
[15] Robson, J.M.: The emptiness of complement problem for semi extended regular expressions requires c^n space. Inform. Processing Letters 9(5), 220–222 (1979)