Bit-coded Regular Expression Parsing

{ Lasse Nielsen, Fritz Henglein }**

DIKU, University of Copenhagen

Abstract. Regular expression parsing is the problem of producing a parse tree of a string for a given regular expression. We show that a compact bit representation of a parse tree can be produced efficiently, in time linear in the product of input string size and regular expression size, by simplifying the DFA-based parsing algorithm due to Dubé and Feeley to emit the bits of the bit representation without explicitly materializing the parse tree itself. We furthermore show that Frisch and Cardelli's greedy regular expression parsing algorithm can be straightforwardly modified to produce bit codings directly. We implement both solutions as well as a backtracking parser and perform benchmark experiments to gauge their practical performance. We observe that our DFA-based solution can be significantly more time and space efficient than the Frisch-Cardelli algorithm due to its sharing of DFA-nodes, but that the latter may still perform better on regular expressions that are "more deterministic" from the right than the left. (Backtracking is, unsurprisingly, quite hopeless.)

1 Introduction

A *regular expression* over finite alphabet Σ, as introduced by Kleene [12], is a formal expression generated by the regular tree grammar

$$E, F ::= 0 \mid 1 \mid a \mid E + F \mid E \times F \mid E^*$$

where $a \in \Sigma$, and *, $\times$ and $+$ have decreasing precedence. (In concrete syntax, we may omit $\times$ and write | instead of +.) Informally, we talk about a regular expression *matching* a string, but what exactly does that mean?

In classical theoretical computer science, regular expression matching is the problem of *deciding* whether a string belongs to the regular *language* denoted by a regular expression; that is, it is *membership testing* [1]. In this sense, *abdabc* matches `((ab)(c|d)|(abc))*`, but *abdabb* does not. This is captured in the *language interpretation* for regular expressions in Figure 2.1.

In *programming*, however, membership testing is rarely good enough: We do not only want a yes/no answer, we also want to obtain proper *matches* of substrings against the subexpressions of a regular expression so as to *extract* parts of the input for further processing. In a *Perl Compatible Regular Expression (PCRE)*[1] matcher, for example, matching *abdabc* against $E =$ `((ab)(c|d)|(abc))*` yields a substring match for each of the 4 parenthesized subexpressions ("groups"): They

** This work has been partially supported by the Danish Strategic Research Council under Project "TrustCare". The order of authors is insignificant.

[1] See `http://www.pcre.org`.

match abc, ab, c, and ϵ (the empty string), respectively. If we use a POSIX matcher [10] instead, we get $abc, \epsilon, \epsilon, abc$, however. The reason for the difference is that `((ab)(c|d)|(abc))*` is *ambiguous*: the string abc can match the left or the right alternative of `(ab)(c|d)|(abc)`, and returning substring matches makes this difference observable.

A limitation of Perl- and POSIX-style matching is that we only get at most one match for each group in a regular expression. This is why only abc is returned, the *last* substring of $abdabc$ matching the group `((ab)(c|d)|(abc))` in the regular expression above. Intuitively, we might expect to get the *list* of all matches $[abd, abc]$. This is possible with *regular expression types* [9]: Each group in a regular expression can be named by a variable, and the output may contain multiple matches for the same variable. For a variable under *two* Kleene stars, however, we cannot discern the matches belonging to different level-1 Kleene-star groups.

An even more refined notion of matching is thus *regular expression parsing*: Returning a parse tree of the input string under the regular expression read as a *grammar*. Automata-theoretic techniques, which exploit equivalence of regular expressions under their language interpretation, typically change the grammatical structure of matched strings and are thus not directly applicable. Only recently have linear-time[2] regular expression *parsing* algorithms been devised [6, 7].

In this paper we show how to generate a compact *bit-coded* representation of a parse tree highly efficiently, without explicitly constructing the parse tree first. A bit coding can be thought of as an oracle directing the expansion of a grammar—here we only consider regular expressions—to a particular string. Bit codings are interesting in their own right since they are typically not only smaller than the parse tree, but also smaller than the string being parsed and can be combined with other techniques for improved text compression [4, 3].

In Section 2 we recall that parse trees can be identified with the elements of regular expressions interpreted as types, and in Section 3 we describe bit codings and conversions to and from parse trees. Section 4 presents our algorithms for generating bit coded parse trees. These are empirically evaluated in Section 5. Section 6 summarizes our conclusions.

2 Regular Expressions as Types

Parse trees for regular expressions can be formalized as ad-hoc data structures [6, 2], representing exactly how the string can be expressed in the regular expression. This means that both membership testing, substring groups and regular expression types can be found by filtering away the extra information in the parse tree. Interestingly, parse trees also arise completely naturally by interpreting regular expressions as *types* [7, 8]; see Figure 2.2. For example, the type interpretation of regular expression $((\mathtt{ab})(\mathtt{c}|\mathtt{d})|(\mathtt{abc}))*$ is $((\{a\} \times \{b\}) \times (\{c\} + \{d\}) + \{a\} \times (\{b\} \times \{c\}))\,\mathrm{list}$. We call elements of a type *values*; e.g., $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inr}\,(a, (b, c))]$ and $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\,d), \mathrm{inl}\,((a, b), \mathrm{inl}\,c)]$ are different elements of $((\{a\}\times\{b\})\times(\{c\}+$

Fig. 2.1 The language interpreation of regular expressions

$\mathcal{L}[\![0]\!] = \emptyset$	$\mathcal{L}[\![1]\!] = \{\epsilon\}$	$\mathcal{L}[\![a]\!] = \{a\}$
$\mathcal{L}[\![E + F]\!] = \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!]$	$\mathcal{L}[\![E \times F]\!] = \mathcal{L}[\![E]\!]\, \mathcal{L}[\![F]\!]$	$\mathcal{L}[\![E^*]\!] = \{\bigcup_{i \geq 0} \mathcal{L}[\![E]\!]^i$

where ϵ is the empty string, $S\,T = \{s\,t \mid s \in S \wedge t \in T\}$, and $S^0 = \{\epsilon\}, S^{i+1} = S\,S^i$.

Fig. 2.2 The type interpreation of regular expressions

$\mathcal{T}[\![0]\!] = \emptyset$	$\mathcal{T}[\![1]\!] = \{()\}$	$\mathcal{T}[\![a]\!] = \{a\}$
$\mathcal{T}[\![E + F]\!] = \mathcal{T}[\![E]\!] + \mathcal{T}[\![F]\!]$	$\mathcal{T}[\![E \times F]\!] = \mathcal{T}[\![E]\!] \times \mathcal{T}[\![F]\!]$	$\mathcal{T}[\![E^*]\!] = \mathcal{T}[\![E]\!]\,\mathrm{list}$

where $()$ is distinct from all alphabet symbols, $S + T = \{\mathrm{inl}\,x \mid x \in S\} \cup \{\mathrm{inr}\,y \mid y \in T\}$ is the disjoint union, $S \times T = \{(x, y) \mid x \in S, y \in T\}$ the Cartesian product of S and T, and $S\,\mathrm{list} = \{[v_1, \ldots, v_n] \mid v_i \in S\}$ the finite lists over S.

$\{d\}) + \{a\} \times (\{b\} \times \{c\}))\,\mathrm{list}$ and thus represent different parse trees for regular expression `((ab)(c|d)|(abc))*`.

We can *flatten* (unparse) any value to a string by removing the tree structure.

Definition 2.1. *The* flattening *function* flat(.) *from values to strings is defined as follows:*

$$\begin{array}{rlrl}
\mathrm{flat}(()) = \epsilon & & \mathrm{flat}(a) = a \\
\mathrm{flat}(\mathrm{inl}\,v) = \mathrm{flat}(v) & & \mathrm{flat}(\mathrm{inr}\,w) = \mathrm{flat}(w) \\
\mathrm{flat}((v, w)) = \mathrm{flat}(v)\,\mathrm{flat}(w) & & \mathrm{flat}(\mathrm{fold}\,v) = \mathrm{flat}(v)
\end{array}$$

Flattening the type interpretation of a regular expression yields its language interpretation:

Theorem 2.1. $\mathcal{L}[\![E]\!] = \{\mathrm{flat}(v) \mid v \in \mathcal{T}[\![E]\!]\}$

A regular expression is *ambiguous* if and only if its type interpretation contains two distinct values that flatten to the same string. With p_1, p_2 as above, since $\mathrm{flat}(p_1) = \mathrm{flat}(p_2) = abdabc$, this shows that $((ab)(c|d)|(abc))^*$ is grammatically ambiguous.

3 Bit-coded Parse Trees

The description of bit coding in this section is an adaptation from Henglein and Nielsen [8]. Bit coding for more general types than the type interpretation of regular expressions is well-known [11]. It has been applied to certain context-free grammars [3], but its use in this paper for efficient regular expression parsing seems to be new.

A *bit-coded parse tree* is a bit sequence representing a parse tree for a *given* regular expression. Intuitively, bit coding factors a parse tree p into its static part, the regular expression E it is a member of, and its dynamic part, a bit sequence that uniquely identifies p as a particular element of E. The basic idea is that the bit sequence serves as an oracle for the alternatives that must be taken to expand a regular expression into a particular string.

[2] This is the data complexity; that is for a fixed regular expression, whose size is considered constant.

Fig. 3.1 Type-directed encoding function from syntax trees to bit sequences

$$
\begin{array}{ll}
\mathrm{code}(() : 1) & = \epsilon \\
\mathrm{code}(a : a) & = \epsilon \\
\mathrm{code}(\mathrm{inl}\, v : E + F) & = 0\ \mathrm{code}(v : E) \\
\mathrm{code}(\mathrm{inr}\, w : E + F) & = 1\ \mathrm{code}(w : F) \\
\mathrm{code}((v, w) : E \times F) & = \mathrm{code}(v : E)\ \mathrm{code}(w : F) \\
\mathrm{code}([v_1, \ldots, v_n] : E^*) & = 0\ \mathrm{code}(v_1 : E) \ldots\ 0\ \mathrm{code}(v_n : E)\ 1
\end{array}
$$

Fig. 3.2 Type-directed decoding function from bit sequences to syntax trees

$$
\begin{array}{ll}
\mathrm{decode}'(d : 1) & = ((), d) \\
\mathrm{decode}'(d : a) & = (a, d) \\
\mathrm{decode}'(0d : E + F) & = \textbf{let}\ (v, d') = \mathrm{decode}'(d : E) \\
 & \quad\ \textbf{in}\ (\mathrm{inl}\, v, d') \\
\mathrm{decode}'(1d : E + F) & = \textbf{let}\ (w, d') = \mathrm{decode}'(d : F) \\
 & \quad\ \textbf{in}\ (\mathrm{inr}\, w, d') \\
\mathrm{decode}'(d : E \times F) & = \textbf{let}\ (v, d') = \mathrm{decode}'(d : E) \\
 & \qquad\quad (w, d'') = \mathrm{decode}'(d' : F) \\
 & \quad\ \textbf{in}\ ((v, w), d'') \\
\mathrm{decode}'(0d : E^*) & = \textbf{let}\ (v_1, d') = \mathrm{decode}'(d : E) \\
 & \qquad\quad (\vec{v}, d'') = \mathrm{decode}'(d' : E^*) \\
 & \quad\ \textbf{in}\ (v_1 :: \vec{v}, d') \\
\mathrm{decode}'(1d : E^*) & = ([], d) \\
\mathrm{decode}(d : E) = & = \textbf{let}\ (w, d') = \mathrm{decode}'(d : E) \\
 & \quad\ \textbf{in}\ \textbf{if}\ d' = \epsilon\ \textbf{then}\ w\ \textbf{else}\ \mathit{error}
\end{array}
$$

Consider, for example, the values $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]$ and $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]$ from Section 2, which represent distinct parse trees of *abdabc* for regular expression `((ab)(c|d)|(abc))*`. The bit coding arises from throwing away everything in the parse tree except the list and the tag constructors, which yields $[\mathrm{inl}\,\mathrm{inr}\,, \mathrm{inr}\,]$. We code inl by 0 and inr by 1, which gives us $[01, 1]$. Finally we code the list itself: Each element is prefixed by 0, and the list is terminated by 1. The resulting bit coding is $b_1 = 001\,01\,1$ (whitespace added for readability). Similarly, the bit coding of p_2 is $b_2 = 001\,000\,1$. More compact codings for lists are possible by generalizing regular expressions to tail-recursive μ-terms [8]. We stick to the given coding of lists here, however, since the focus of this paper is on constructing the bit codings, not their effect on text compression.

Figures 3.1 and 3.2 define regular-expression directed linear-time coding and decoding functions from parse trees to their bit codings and back:

Theorem 3.1. *If* $v \in \mathcal{T}[\![E]\!]$ *then* $\mathrm{decode}(\mathrm{code}(v : E) : E) = v$

Proof: By structural induction on v.

Note that bit codings are only meaningful in the context of a regular expression,

because the same bit sequence may represent different strings for different regular expressions, and may be invalid for other regular expressions.

Bit codings are not only more compact than parse trees. As we shall see, they are also more suitable for automaton output, as it is not necessary to generate list structure, pairing or even the alphabet symbols occurring in a parse tree.

4 Parsing Algorithms

We present two bit coding parsing algorithms in this section. The first can be understood as a simplification of Dubé and Feeley's [6] DFA-generation algorithm, producing bit codings instead of explicit parse tree. We also show that Frisch and Cardelli's [7] algorithm can be straightforwardly modified to produce bit codings.

4.1 Dubé/Feeley-style Parsing

Our algorithm DFA performs the following steps: Given regular expression E and input string s,

1. generate an enhanced Thompson-style NFA with output actions (finite state transducer);
2. use the subset construction to produce an enhanced DFA with additional information on edges to capture the output actions from the NFA;
3. use the enhanced DFA as a regular DFA on s;
   - if it rejects terminate with error (no parse tree);
   - if it accepts, return the path in the DFA induced by the input string;
4. combine, in reverse order of the path, the output information on each edge traversed to construct the bit coding of a parse tree.

The steps are described in more detail below.

Enhanced NFA generation. The left column of Figure 4.1 shows Thompson-style NFA generation. We enhance it by adding single bit outputs to the outedges of those states that have two outgoing ϵ-transitions, shown in the right column. The output bits can be thought of indicators for an agent traversing the NFA: 0 means turn left, 1 means turn right. The other edges carry no output bit since their traversal is forced.

When traversing a path p in an enhanced NFA, the sequence of symbols read is denoted by $\mathrm{read}(p)$, and the sequence of symbols written is denoted by $\mathrm{write}(p)$.

Lemma 4.1 (Soundness and completeness of enhanced NFAs). *Let N_E be the extended NFA for regular expression E according to Figure 4.1 (right). Then for each $s \in \Sigma^*$ we have* $\{v | v \in \mathcal{T}[\![E]\!] \wedge \mathrm{flat}(v) = s\} =$
$\{\mathrm{decode}(\mathrm{write}(p) : E) \mid p$ *is a path in N_E from initial state to final state such that* $\mathrm{read}(p) = s\}$.

Proof: By structural induction on E.

In other words, an extended NFA generates exactly the bit codings of the parse trees of the strings it accepts. Observe furthermore that no two distinct paths

from initial to final state have the same output bits. This means that a bit coding uniquely determines a particular path from initial to final state, and vice versa. Dubé and Feeley [6] also instrument Thompson-style NFAs, but with more output symbols on more edges so as to be able to generate an external representation of a parse tree. Figure 4.2 shows their enhanced NFA for $E = a \times (b + c)^\star \times a$ on the left. Our corresponding enhanced NFA is shown on the right.

Enhanced subset construction. During subset construction additional information is computed:

1. A map init from the NFA states q in the initial DFA state to an output $\mathrm{init}(q)$. The output $\mathrm{init}(q)$ must be $\mathrm{write}(p)$ for some path p from the initial NFA state to q where $\mathrm{read}(p) = \varepsilon$. These paths are traversed when finding the ε-closure of the initial NFA state, which is how the initial DFA state is constructed in the subset construction, and is thus easy to generate.
2. A map output_e for each edge e in the DFA, that maps each NFA state q_2 in the destination DFA state to a pair (q_1, o) of an NFA state q_1 in the source DFA state, and an output o. The output o must be $\mathrm{write}(p)$ for some path p from q_1 to q_2 where $\mathrm{read}(p)$ is the input of the DFA edge e. These paths are traversed, when the destination state of the edge is computed, and are thus simple to generate.

The DFA for the NFA from Fig. 4.2 (right) is shown in Fig. 4.3, and the result of adding the information described above is shown in Fig. 4.4.

The additional information captures basically the same information as in Dubé and Feeley's DFA construction, but it stores the additional information directly in the DFA edges, where Dubé and Feeley use an external 3-dimensional table. Most importantly, the additional information we need to store is reduced, since we only generate bit codings, not explicit parse trees.

Bit code construction. After accepting $s = a_1 \ldots a_n$ we have a path $p = [A_0, A_1, \cdots, A_n]$ in the extended DFA, where $A_0, \ldots, A_n$ are DFA states, each consisting of a set of NFA states, with A_0 containing the initial NFA state q_i and A_n the final NFA state q_f.

We construct the bit code of a parse tree for s by calling $\mathrm{write}(p, q_f)$ where write traverses p from right to left as follows:

$$\begin{aligned}
\mathrm{write}([A_0], q) &= \mathrm{init}_q \\
\mathrm{write}([A_0, A_1, \ldots, A_{k-1}, A_k], q) &= \mathrm{write}([A_0, A_1, \ldots, A_{k-1}], q') \cdot b' \\
&\quad \text{where } (q', b') = \mathrm{output}_{A_{k-1} \to A_k}(q)
\end{aligned}$$

Lemma 4.2 (Bit coding preservation). *Let D_E be the extended DFA generated from the extended NFA N_E for E. If p is a path from the initial state in D to a state containing the NFA state q and if* $\mathrm{write}(p, q) = b$
then there is a path p' in N_E from the initial state in N_E to the state q such that $\mathrm{read}(p) = \mathrm{read}(p')$ *and* $\mathrm{write}(p, q) = \mathrm{write}(p')$.

PROOF: Induction on the number of steps in p.

We can now conclude that the bit sequence found represents a parse tree for the input string.

Theorem 4.1 (Correctness of DFA algorithm). *If D_E is an extended DFA generated from an extended NFA N_E for regular expression E, and p is a path from the initial state to a final state in D_E, and q_f is the final state of N_E then* $\mathrm{read}(p) = \mathrm{flat}(\mathrm{decode}(\mathrm{write}(p, q_f) : E))$.

Proof: This follows from first applying Lemma 4.2 and then Lemma 4.1.

Once the DFA has been generated, this method results in very efficient regular expression parsing. The DFA traversal takes time $\Theta(|s|)$, and the bit code generation takes time $\Theta(|s|+|b|)$, where $|b|$ is the length of the output bit sequence. This means the total run time complexity of parsing is $\Theta(|s| + |b|)$ which is (sequentially) optimal, since the entire string must be read, and the entire bit sequence must be written.

Example. Consider the extended DFA for the regular expression $a(b+c)^{\star}a$ in Fig. 4.4.
If we use it to accept the string $abcba$, then we get the path $p = 0 \to 1 \to 3 \to 4 \to 3 \to 2$. Tracing the path backwards, keeping track of the output bit-sequences b and NFA states q we get the steps in the table on the right.

Step	q	Symbol	new q	b
$3 \to 2$	9	a	8	""
$4 \to 3$	8	b	3	"1"
$3 \to 4$	3	c	4	"00"
$1 \to 3$	4	b	3	"01"
$0 \to 1$	3	a	0	"00"

Since $\mathrm{init}_0 =$ "" we get the bit code $b =$ "" $\cdot$ "00" $\cdot$ "01" $\cdot$ "00" $\cdot$ "1" $\cdot$ "" $=$ "0001001".

We can verify that the DFA parsing algorithm has given us a correct bit coding since $\mathrm{flat}(\mathrm{decode}(\text{"0001001"} : a(b+c)^{\star}a)) = abcba$.

4.2 Frisch/Cardelli-style Parsing

Instead of building a Thompson-style NFA from the regular expressions, Frisch and Cardelli [7] build an NFA with one node for each position in the regular expression, with the final state as the only additional node. The regular expression positions are identified by the path used to reach the position. The positions $\lambda_{\mathbf{end}}(E)$ in a regular expression E are defined as lists of choices (the choices are **fst** and **snd** for sequence, **lft** and **rgt** for sum and **star** for $\star$). $E.l$ is used to denote the subexpression of E found by following the path l. The transitions $\delta(E)$ in the NFA of a regular expression E are defined using a successor relation **succ** on the paths.

The NFA is used to generate a table $Q(l,i)$, which maps each position $l \in \lambda_{\mathbf{end}}(E)$ and input string position i to `true`, if l accepts the ith suffix of s and `false` otherwise.

The table Q can be constructed by starting with $Q(l,i) =$ `false` for all $l \in \lambda_{\mathbf{end}}(E)$ and $i = 1 \ldots |s|$, and calling $\mathbf{SetPrefix}(Q, \mathbf{end}, |s|)$, which updates Q as defined below.

$\mathbf{SetPrefix}(Q,l,i) =$ if $Q(l,i)$ then return;
$Q(l,i) :=$ `true`;
for $(l', \varepsilon, l) \in \delta(E)$ do $\mathbf{SetPrefix}(Q, l', i)$;
if $i > 0$ then for $(l', s[i], l) \in \delta(E)$ do $\mathbf{SetPrefix}(Q, l', i-1)$;

Fig. 4.1 NFA generation schema

E	NFA	Extended NFA
0	0 1	0 1
1	0	0
a	0 a 1	0 a/ε 1
$E_1 \times E_2$	0 E 1 F 2	0 E 1 F 2
$E + F$	0 1 E_1 3 2 E_2 4 5	0 ε/1 1 E_1 3 ε/ε ε/0 2 E_2 4 ε/ε 5
$E_1{}^\star$	1 E_1 2 0 3	1 E_1 2 ε/ε ε/0 ε/1 0 3

Fig. 4.2 Extended NFAs for $a(b+c)^\star a$

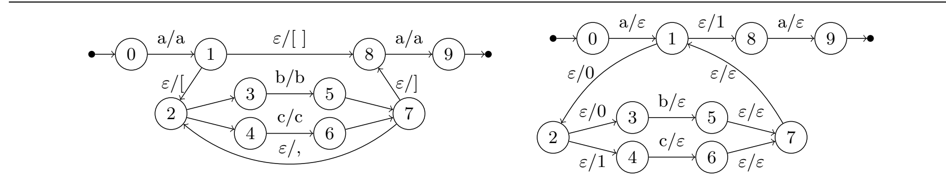


Fig. 4.3 DFA for $a(b+c)^\star a$

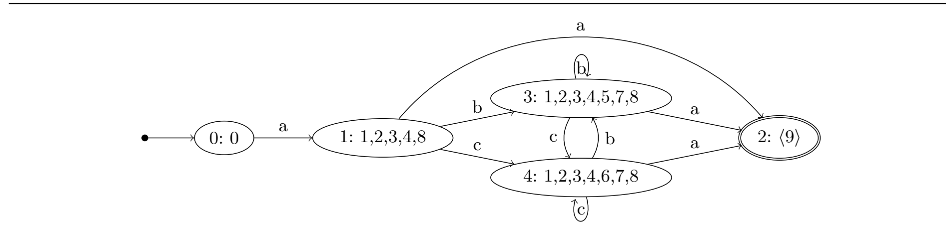
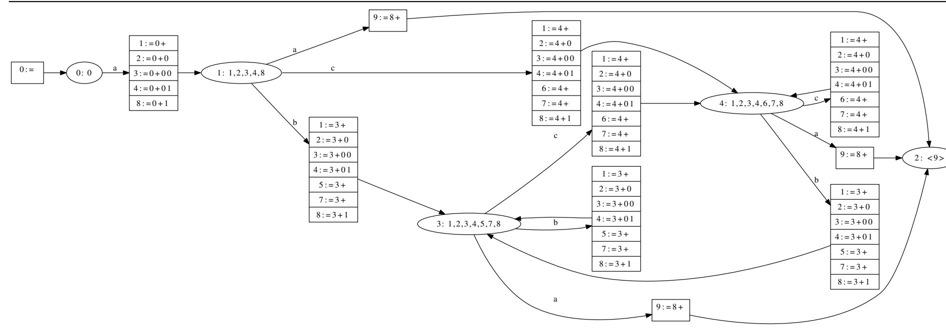


Fig. 4.4 Extended DFA for $a(b+c)^\star a$



The run time cost and memory consumption of computing Q is asymptotically bounded by the size of Q, which is in $\Theta(|s| \cdot |E|)$.

After Q is built, it is easy to check whether s matches the regular expression E, simply by looking up $Q([], 0)$. We modify Frisch and Cardelli's build function to construct a bit coding representing the greedy leftmost (or first and greedy [14]) parse tree for s, if s matches E, as follows (notice that $l :: x$ means appending x after l):

$$
\begin{aligned}
\mathbf{build}(l, i) = \text{case } E.l \text{ of} \\
& a\text{: return } (\varepsilon, i+1) \\
& 1\text{: return } (\varepsilon, i) \\
& E_1 \times E_2\text{: let } (b_1, j) = \mathbf{build}(l :: \mathbf{fst}, i) \\
& \qquad \text{in let } (b_2, k) = \mathbf{build}(l :: \mathbf{snd}, j) \text{ in return } (b_1 b_2, k); \\
& E_1 + E_2\text{: if } Q(l :: \mathbf{lft}, i) \\
& \qquad \text{then let } (b_1, j) = \mathbf{build}(l :: \mathbf{fst}, i) \text{ in retuen } (0 b_1, j) \\
& \qquad \text{else let } (b_2, j) = \mathbf{build}(l :: \mathbf{snd}, i) \text{ in return } (1 b_2, j); \\
& E_1^{\star}\text{: if } Q(l :: \mathbf{star}, i) \\
& \quad \text{then let } (b_1, j) = \mathbf{build}(l :: \mathbf{star}, i) \\
& \qquad \text{in let } (b_2, k) = \mathbf{build}(l, j) \text{ in return } (0b1b2, k) \\
& \quad \text{else return } (1, j);
\end{aligned}
$$

The run time cost and memory consumption of $\mathbf{build}([], 0)$ is asymptotically bounded by the number of cells in Q, which is in $\Theta(|E| \cdot |s|)$ and which is therefore the time complexity and memory consumption of the entire algorithm.

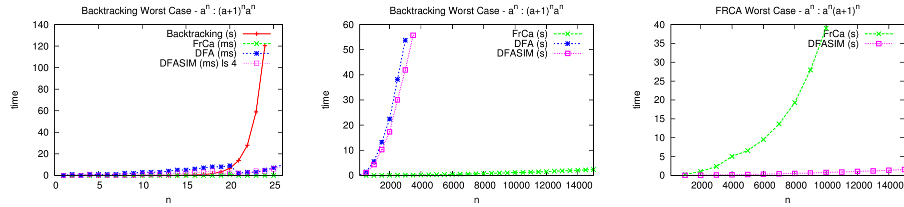
5 Empirical evaluation

We have implemented the algorithms described in Section 4 as a `C++` library [13] and performed a series of performance tests on a PC with a 2.50GHz Intel Core2 Duo CPU and 4Gb of memory, running Ubuntu 10.4. We test four different parsing methods. NFA based backtracking (backtracking), implemented by a depth-first search for an accepting path in our enhanced Thompson-style NFA. FRCA is the algorithm based on Frisch and Cardelli from Section 4.2. DFA is the algorithm based on Dubé and Feeley from Section 4.1. DFASIM is the same algorithm as DFA, but where the nodes and edges of the DFA are not precomputed, but generated dynamically by need.

5.1 Backtracking Worst Case: ($\mathtt{a}^n$: $(a+1)^n a^n$)

The regular expression is $(a+1)^n a^n$, where we use the notation E^n to represent $E \times \cdots \times E$ (n copies). This is a well-known example [5], which captures the problematic cases for backtracking.[3] The results of matching $\mathtt{a}^n$ (denoting n `a`s) to $(a+1)^n \times a^n$ are in the two leftmost graphs below.

[3] If a fixed regular expression is preferred, then $(a+a)^{\star} \times b$ or $(a^{\star} \times a)^{\star} \times b$ provokes the same behavior.

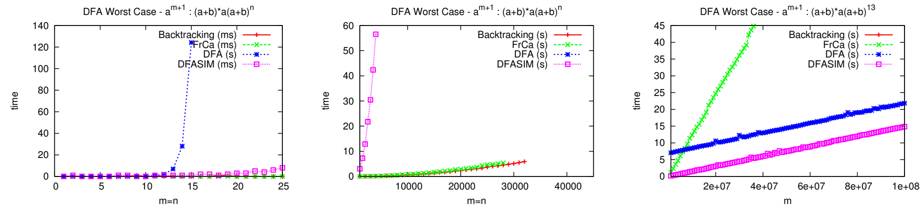


When parsing a string with n as, the backtracking algorithm traverses 2^n different paths before eventually finding the match. The cost of generating the DFA is $\Theta(n^2)$ ($2 \cdot n$ nodes containing n NFA-nodes on average). Since only half of the DFA-nodes are used, DFASIM is faster than generating the whole DFA, but the run time complexity is still $\Theta(n^2)$. The time used for FRCA is $\Theta(n)$, since there is exactly one suffix that can be parsed from each position in the regular expression. The reason FRCA performs much better than the other algorithms in this example is that the example was designed to be hard to parse from the left to right, and FRCA processes the string in its first phase from right to left. If we change the regular expression to $a^n(1+a)^n$, it becomes hard to process from right to left, as shown in the rightmost graph above. It is generally advantageous to process a string in the "more deterministic" direction of the regular expression.

5.2 DFA Worst Case ($\mathtt{a}^{m+1} : (a+b)^\star a(a+b)^n$)

The following is a worst-case scenario for the DFA based algorithm, and a best-case scenario for the FRCA and backtracking algorithms. The regular expression is $(a+b)^\star a(a+b)^n$, and the string is $\mathtt{a}^{m+1}$.

The two leftmost graphs below show the execution time when $n = m$, and the right graph shows the execution time when $n = 13$. When n is fixed to 13, the runtime of both FRCA, DFASIM and DFA are linear, but even though DFA has a large initialization time for building the DFA, FRCA uses more time for large m, because it uses more time per character.

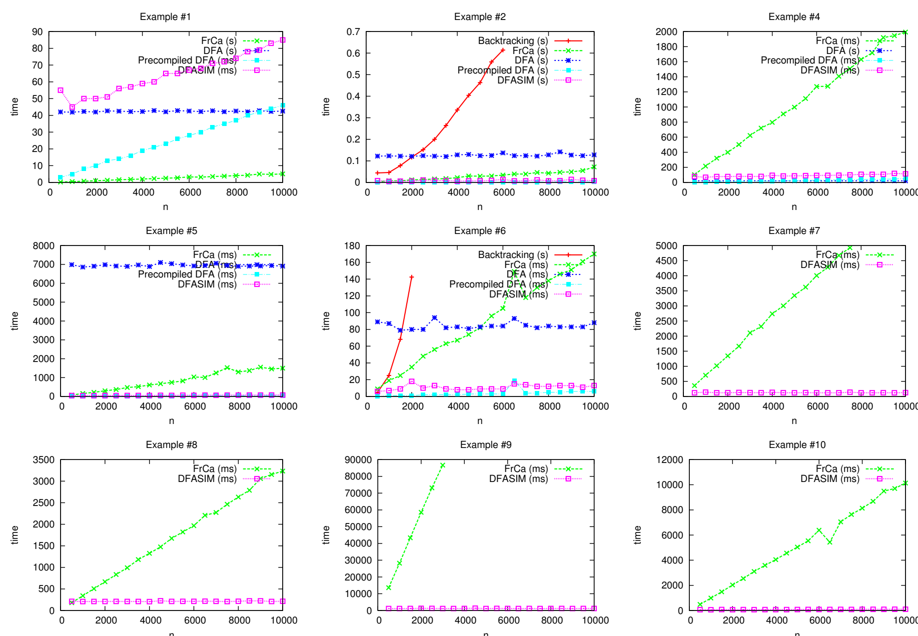


The DFA will have $\mathcal{O}(2^n)$ DFA-nodes. This causes the DFA algorithm to have a run time complexity of $\Theta(m \cdot 2^n)$. This exponential explosion is avoided by DFASIM, which only builds as many states as needed for the particular input string.

5.3 Practical examples

We have tested 9 of the 10 examples of "real world" regular expressions from Veanes et al. [15] to compare the performance of each algorithm. (Their example nr. 3 is uninteresting for performance testing since it only accepts strings of a bounded length).

Examples 1,4,5,7,8,9,10 are different ways of expressing the language of email addresses, while Example 2 defines the language of dollar-amounts, and Example 6 defines the language of floating point values.



The DFA and Precompiled DFA (a staged version of DFA) graphs are missing in many of the examples. This is because the DFA generation runs out of memory. We may conclude that there are many cases where it is not feasible to generate the full DFA. The two best algorithms for these tests are FRCA and DFASIM, with DFASIM being faster by a large factor (at least 10) in all cases. Apart from the direction of processing NFA-nodes in their respective first passes, the key difference between DFASIM and FRCA is that DFASIM memoizes and reuses DFA-states at multiple positions in the input string, whereas FRCA essentially produces what amounts to a separate DFA-state for each position in the input string. In comparison to FRCA, the DFA-state memoization not only saves space, but also computation time whenever the same transition is traversed more than once.

6 Conclusion

We have designed and implemented a number of regular expression parsing algorithms, which produce bit coded representations of parse trees without ever materializing the parse trees during parsing. Producing bit codings is advantageous since it carries the dual advantage of yielding a compressed parse tree representation and of speeding its construction. Our DFA simulation algorithm DFASIM, in the style of Dubé and Feeley [6], and FRCA, a modified version of the greedy algorithm of Frisch and Cardelli [7], have shown the best asymptotic performance,

with DFA simulation beating FRCA on a suite of real world examples. As for the potential for further improvements, compact NFA-construction, efficient computation of the sub-NFA induced by an input string (left-to-right or right-to-left or, preferably, something better), and memoized DFA-state construction appear to be key to obtaining practically improved regular expression parsing without sacrificing asymptotic scalability.

References

1. P. Bille and M. Thorup. Faster regular expression matching. *Automata, Languages and Programming*, pages 171–182, 2009.
2. C. Brabrand and J. Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. In *Proc. 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)*, 2010.
3. R. Cameron. Source encoding using syntactic information source models. *Information Theory, IEEE Transactions on*, 34(4):843–850, 1988.
4. J. Contla. Compact coding of syntactically correct source programs. *Software: Practice and Experience*, 15(7):625–636, 1985.
5. R. Cox. Regular expression matching can be simple and fast.
6. D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, September 2000.
7. A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture notes in computer science*, pages 618–629, Turku, Finland, July 2004. Springer.
8. F. Henglein and L. Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation (preliminary version). In *Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2011.
9. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.
10. Institute of Electrical and Electronics Engineers (IEEE). *Standard for information technology — Portable Operating System Interface (POSIX) — Part 2 (Shell and utilities), Section 2.8 (Regular expression notation)*. New York, 1992. IEEE Standard 1003.2.
11. P. Jansson and J. Jeuring. Polytypic compact printing and parsing. *Programming Languages and Systems*, pages 639–639, 1999.
12. S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata studies*, 34:3–41, 1956.
13. L. Nielsen. Regular expression compression parser.
Online: http://www.thelas.dk/index.php/Rcp.
14. S. Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006.
15. M. V. M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proc. 3d Int'l Conf. on Software Testing, Verification and Validation*, Paris, France, April 6-10 2010. IEEE Computer Society Press.