Introduction to Regular Expression Derivatives

A functional, algebraic approach to generate DFAs

Schroedinger Yifan ZHU

Computer Science Department
University of Rochester
Agenda

- Derivative-based DFA generation for individual regular expressions
  - Regular expression definitions (set-based)
  - Derivatives of REs
  - Weak-equivalence between REs
  - Congruence classes of the alphabet
- Derivative-based DFA generation for lexers (groups of REs)
What are regular expressions?

Regular expressions on an ambient set $\Sigma$ (a.k.a. alphabet) is inductively defined with the following rules:

$$\mathcal{L} := \mathcal{I} \mid \varepsilon \mid \alpha \sim \beta \mid \alpha + \beta \mid \alpha \& \beta \mid \alpha^* \mid \neg \alpha \mid \emptyset$$  \hspace{1cm} (1)

- We may abuse single characters (‘a’) to represent singleton sets.
- We use some pictures from Scott Owens’ Regular-expression derivatives re-examined, where $\sim$ may be omitted or replaced by $\cdot$ instead.
What are regular expressions?

Regular expressions on an ambient set $\Sigma$ (a.k.a. alphabet) is inductively defined with the following rules:

- $\mathcal{S} \subseteq \Sigma$: single character languages where the character comes from a subset of the alphabet.
- $\epsilon$: empty languages.
- $\alpha \sim \beta$: languages where sub-language $\alpha$ immediately followed by sub-language $\beta$.
- $\alpha + \beta$: union of languages represented by $\alpha$ or $\beta$.
- $\alpha \& \beta$: intersection of languages represented by $\alpha$ and $\beta$.
- $\alpha^*$: Kleene closure of $\alpha$.
- $\emptyset$: no language.
- $\neg \alpha$: languages that $\alpha$ rejects.
What are regular expression derivatives?

A **derivative** of a regular expression \( L \) over a character is another regular expression \( L' \) that accepts languages that are accepted in \( L \) after character is consumed.

\[
\partial_a(a \sim b \sim c) = b \sim c \\
\partial_b(a \sim b \sim c) = \emptyset \\
\partial_a((a + b) \sim b \sim c) = b \sim c
\]
What are regular expression derivatives?

\[ \partial_\gamma (I) = \begin{cases} \varepsilon, & \gamma \in I \\ \emptyset, & \gamma \notin I \end{cases} \]

\[ \partial_\gamma (\varepsilon) = \emptyset \]

\[ \partial_\gamma (\alpha \sim \beta) = \begin{cases} (\partial_\gamma (\alpha) \sim \beta) + \partial_\gamma (\beta), & \text{nullable}(\alpha) \\ \partial_\gamma (\alpha) \sim \beta, & \neg \text{nullable}(\alpha) \end{cases} \]

\[ \partial_\gamma (\alpha + \beta) = \partial_\gamma (\alpha) + \partial_\gamma (\beta) \]

\[ \partial_\gamma (\alpha \& \beta) = \partial_\gamma (\alpha) \& \partial_\gamma (\beta) \]

\[ \partial_\gamma (\alpha^*) = \partial_\gamma (\alpha) \sim \alpha^* \]

\[ \partial_\gamma (\neg \alpha) = \neg \partial_\gamma (\alpha) \]
What are regular expression derivatives?

\[
\begin{align*}
\text{nullable } (S) &= \text{False} \\
\text{nullable } (\emptyset) &= \text{False} \\
\text{nullable } (\epsilon) &= \text{True} \\
\text{nullable } (\alpha \sim \beta) &= \text{nullable } (\alpha) \land \text{nullable } (\beta) \\
\text{nullable } (\alpha + \beta) &= \text{nullable } (\alpha) \lor \text{nullable } (\beta) \\
\text{nullable } (\alpha \& \beta) &= \text{nullable } (\alpha) \land \text{nullable } (\beta) \\
\text{nullable } (\alpha^*) &= \text{True} \\
\text{nullable } (\neg \alpha) &= \neg \text{nullable } (\alpha)
\end{align*}
\]
The big picture

- Each regular expression $\mathcal{L}$ represents a state in DFA.
- At each state $\mathcal{L}$, enumerate the alphabet $\Sigma$ and create an edge from $\mathcal{L}$ to $\partial_\gamma(\mathcal{L})$ for character $\gamma$.
- Successful matches are represented by \textbf{nullable} expressions.
- Rejecting states are indicated by $\emptyset$. 
The big picture

Figure: An example DFA for $(a \sim b) + (a \sim c)$
Semantically identical regular expressions?

For regular expressions with intersection and complement, checking if two expressions accept the same set of languages may acquire **non-elementary time complexity** (Aho, et al. The Design and Analysis of Computer Algorithms)
A weaker notion of equivalence

\[
\begin{align*}
    r & \& r & \approx & r \\
    r & \& s & \approx & s & \& r \\
    (r & s) & \& t & \approx & r & \& (s & t) \\
    \emptyset & \& r & \approx & \emptyset \\
    \neg \emptyset & \& r & \approx & r \\
    (r \cdot s) \cdot t & \approx & r \cdot (s \cdot t) \\
    \emptyset \cdot r & \approx & \emptyset \\
    r \cdot \emptyset & \approx & \emptyset \\
    \varepsilon \cdot r & \approx & r \\
    r \cdot \varepsilon & \approx & r
\end{align*}
\]  

\[
\begin{align*}
    r + r & \approx r \\
    r + s & \approx s + r \\
    (r + s) + t & \approx r + (s + t) \\
    \neg \emptyset + r & \approx \neg \emptyset \\
    \emptyset + r & \approx r \\
    (r^*)^* & \approx r^* \\
    \varepsilon^* & \approx \varepsilon \\
    \emptyset^* & \approx \varepsilon \\
    \neg (\neg r) & \approx r
\end{align*}
\]

Figure: A weaker notion of equivalence between REs.
A weaker notion of equivalence

- One can add/remove rules based on practical cases. (DotNet’s alternation rule is not commutative)
- If you see the rule from left to right, it actually provides a way to simplify and reorder rules.
- We use $\text{Wnorm}(L)$ to represent apply weakly normalization on expression $L$ based on this weaker notion of equivalence.
Congruence classes

Iterate through the whole alphabet? Too costly for larger alphabets!
Given two characters $\alpha, \beta \in \Sigma$ and a regular expression $L$ (denoted by $\alpha \simeq_L \beta$), we say $\alpha$ is congruent to $\beta$ at expression $L$ if $\partial_\alpha(L)$ and $\partial_\beta(L)$ accept the same set of languages.

\[
a \simeq_{(a|b)\sim c} b \iff \partial_a((a|b) \sim c) \equiv \partial_b((a|b) \sim c) \equiv c
\]

\[
d \simeq_{(a|b)\sim c} e \iff \partial_d((a|b) \sim c) \equiv \partial_e((a|b) \sim c) \equiv \emptyset
\]
If we know all the congruence classes for $L$, we only need to calculate the derivatives per class.

Still no precise solution within reasonable time complexity.

Finer partitions are safe!
If we know all the congruence classes for $\mathcal{L}$, we only need to calculate the derivatives per class.

Still no precise solution within reasonable time complexity.

Finer partitions are safe!
Congruence classes

\[
\begin{align*}
\text{ApproxCC}(\epsilon) &= \{\Sigma\} \\
\text{ApproxCC}(\mathcal{I}) &= \{\mathcal{I}, \Sigma - \mathcal{I}\} \\
\text{ApproxCC}(\alpha \sim \beta) &= \begin{cases} \\
\text{ApproxCC}(\alpha), & \text{(nullable}(\alpha)} \\
\text{ApproxCC}(\alpha) \land \text{ApproxCC}(\beta), & \text{nonnullable}(\alpha)
\end{cases} \\
\text{ApproxCC}(\alpha + \beta) &= \text{ApproxCC}(\alpha) \land \text{ApproxCC}(\beta) \\
\text{ApproxCC}(\alpha \& \beta) &= \text{ApproxCC}(\alpha) \land \text{ApproxCC}(\beta) \\
\text{ApproxCC}(\alpha^*) &= \text{ApproxCC}(\alpha) \\
\text{ApproxCC}(\lnot \alpha) &= \text{ApproxCC}(\alpha) \\
S \land T &= \{s \cap t \mid s \in S \lor t \in T\}
\end{align*}
\]
Congruence classes

\[ \text{ApproxCC}((a + (b \sim a)) \sim c) \]

\[ = \text{ApproxCC}(a + (b \sim a)) \land \text{ApproxCC}(c) \]

\[ = \text{ApproxCC}(a) \land \text{ApproxCC}(b \sim a) \land \text{ApproxCC}(c) \]

\[ = \text{ApproxCC}(a) \land \text{ApproxCC}(b) \land \text{ApproxCC}(c) \]

\[ = \{a, \Sigma - a\} \land \{b, \Sigma - b\} \land \{c, \Sigma - c\} \]

\[ = \{\emptyset, a, b, \Sigma - \{a, b\}\} \land \{c, \Sigma - c\} \]

\[ = \{\emptyset, a, b, c, \Sigma - \{a, b, c\}\} \]

Precise result is \( \{\{a, c\}, b, \Sigma - \{a, b, c\}\} \).
Algorithm 1 DFA states exploration

procedure EXPLORE_DFA(x, V, T) > x is current state (normalized), V is the set of visited states
    if x' \in V then
        return
    end if
    V ← V \cup \{x'\}
    for c ∈ APPROX_CC(x') do
        if IS_EMPTY(c) then
            continue
        end if
        r ← REPRESENTATIVE(c)
        y ← WNORM(\partial_r(x'))
        T[x'][c] ← y
        EXPLORE_DFA(y, V, T)
    end for
end procedure

> update DFA transition table

Figure: DFA States Exploration
A regular expression vector \((\ell_1, \ell_2, \ldots, \ell_n) \in \mathcal{L}^n\) is just a tuple of regular expressions.

\[
\text{Wnorm}\left(\left(\ell_1, \ldots, \ell_n\right)\right) = \left(\text{Wnorm}(\ell_1), \ldots, \text{Wnorm}(\ell_n)\right)
\]

\[
\text{ApproxCC}\left(\left(\ell_1, \ldots, \ell_n\right)\right) = \bigwedge_{i=1}^{n} \text{ApproxCC}(\ell_i)
\]

\[
\partial\left(\left(\ell_1, \ldots, \ell_n\right)\right) = \left(\partial(\ell_1), \ldots, \partial(\ell_n)\right)
\]

(\(\bigwedge\) is the meet operator defined in \text{ApproxCC}.)
Regular expression vectors for lexer generation

\[
\text{Accept} \left( (\ell_1, \ell_2, \ldots, \ell_n) \right) = \bigvee_{i=1}^{n} \text{nullable}(\ell_i)
\]

\[
\text{Reject} \left( (\ell_1, \ell_2, \ldots, \ell_n) \right) = \bigwedge_{i=1}^{n} \ell_i \equiv \emptyset
\]

(\bigwedge, \bigvee \text{ are logical operators.})
Algorithm 2 Longest-match Lexer

procedure LEXER(S) 
    \( M \leftarrow 0 \) \( \quad \) \( \triangleright S \) is the input
    \( L \leftarrow 0 \) \( \quad \) \( \triangleright M \) is the longest match, initialized to 0 (no match)
    \( l \leftarrow \text{LENGTH}(S) \) \( \quad \) \( \triangleright L \) is offset of the longest match
    \( s \leftarrow \text{State}_0 \)
    for \( i \in \text{RANGE}(0, l) \) do 
        \( c \leftarrow S[i] \)
        if \( s = \text{State}_0 \) then 
            if comptime ACCEPT(\text{State}_0) then 
                \( L \leftarrow i \)
                \( M \leftarrow \text{comptime FIRST ACCEPT(}\text{State}_0) \)
            end if 
            if comptime REJECT(\text{State}_0) then 
                break 
            end if 
            if \( c \in \text{comptime APPRXCC(}\text{State}_0)[0] \) then 
                \( s \leftarrow \text{comptime T[State}_0][\text{APPRXCC(}\text{State}_0)[0]] \)
            end if 
        go through all congruence classes 
    end if 
    go through all states 
end for 
return \((M, L)\) 
end procedure
<table>
<thead>
<tr>
<th>Lexer</th>
<th>ml-lex</th>
<th>ml-ulex</th>
<th>Minimal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burg</td>
<td>61</td>
<td>58</td>
<td>58</td>
<td>A tree-pattern match generator</td>
</tr>
<tr>
<td>CKit</td>
<td>122</td>
<td>115</td>
<td>115</td>
<td>ANSI C lexer</td>
</tr>
<tr>
<td>Calc</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>Simple calculator</td>
</tr>
<tr>
<td>CM</td>
<td>153</td>
<td>146</td>
<td>146</td>
<td>The SML/NJ compilation manager</td>
</tr>
<tr>
<td>Expression</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>A simple expression language</td>
</tr>
<tr>
<td>FIG</td>
<td>150</td>
<td>144</td>
<td>144</td>
<td>A foreign-interface generator</td>
</tr>
<tr>
<td>FOL</td>
<td>41</td>
<td>41</td>
<td>41</td>
<td>First-order logic</td>
</tr>
<tr>
<td>HTML</td>
<td>52</td>
<td>49</td>
<td>49</td>
<td>HTML 3.2</td>
</tr>
<tr>
<td>MDL</td>
<td>161</td>
<td>158</td>
<td>158</td>
<td>A machine-description language</td>
</tr>
<tr>
<td>ml-lex</td>
<td>121</td>
<td>116</td>
<td>116</td>
<td>The ml-lex lexer</td>
</tr>
<tr>
<td>Scheme</td>
<td>324</td>
<td>194</td>
<td>194</td>
<td>R₅RS Scheme</td>
</tr>
<tr>
<td>SML</td>
<td>251</td>
<td>244</td>
<td>244</td>
<td>Standard ML lexer</td>
</tr>
<tr>
<td>SML/NJ</td>
<td>169</td>
<td>158</td>
<td>158</td>
<td>SML/NJ lexer</td>
</tr>
<tr>
<td>Pascal</td>
<td>60</td>
<td>55</td>
<td>55</td>
<td>Pascal lexer</td>
</tr>
<tr>
<td>ml-yacc</td>
<td>100</td>
<td>94</td>
<td>94</td>
<td>The ml-yacc lexer</td>
</tr>
<tr>
<td>Russo</td>
<td>4803</td>
<td>3017</td>
<td>2892</td>
<td>System-log data mining</td>
</tr>
<tr>
<td>$L_2$</td>
<td>n/a</td>
<td>147</td>
<td>106</td>
<td>Monitoring stress-test</td>
</tr>
</tbody>
</table>
Preprocessing Time

Wide Spectrum of Regex Matching Techniques

Matching cost

Backtracking

NFA

Derivatives

DFA

Preprocessing cost

Thompson’68 algo ⇒ grep, RE2 (Google), Rust ...
Glushkov’61 algo ⇒ Hyperscan (Intel)
Antimirov’95 derivatives

Brzozowski’64 derivatives ⇒ SRM
On-the-fly determinization of NFA

IsMatch:
Fischer et al. 2010.
Owens et al. 2009.
POSIX semantics:
Sulzmann and Lu. 2012.
Ausaf et al. 2016.

This paper: .NET RegexOptions.NonBacktracking
PCRE semantics: for regexes with anchors

Figure: PLDI slides for Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics