Introduction to Regular Expression Derivatives

A functional, algebraic approach to generate DFAs

Schrodinger 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 Σ (a.k.a. alphabet) is inductively defined with the following rules:

$$\mathscr{L} := \mathscr{S} \mid \varepsilon \mid \alpha \sim \beta \mid \alpha + \beta \mid \alpha \& \beta \mid \alpha^* \mid \neg\alpha \mid \emptyset \tag{1}$$

- 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 Σ (a.k.a. alphabet) is inductively defined with the following rules:

- $\mathscr{S} \subseteq \Sigma$: single character languages where the character comes from a subset of the alphabet.
- ϵ: empty languages.
- $\alpha \sim \beta$: languages where sub-language α immediately followed by sub-language β.
- $\alpha + \beta$: union of languages represented by α or β.
- $\alpha \& \beta$: intersection of languages represented by α and β.
- α^*: Kleene closure of α.
- $\emptyset$: no language.
- $\neg\alpha$: languages that α rejects.

What are regular expression derivatives?

A **derivative** of a regular expression $\mathscr{L}$ over a character is another regular expression $\mathscr{L}'$ that accepts languages that are accepted in $\mathscr{L}$ 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?

$$
\begin{aligned}
\partial_\gamma(\mathscr{S}) &= \begin{cases} \varepsilon, & \gamma \in \mathscr{S} \\ \emptyset, & \gamma \notin \mathscr{S} \end{cases} \\
\partial_\gamma(\varepsilon) &= \emptyset \\
\partial_\gamma(\alpha \sim \beta) &= \begin{cases} (\partial_\gamma(\alpha) \sim \beta) + \partial_\gamma(\beta), & \textbf{nullable}(\alpha) \\ \partial_\gamma(\alpha) \sim \beta, & \neg\textbf{nullable}(\alpha) \end{cases} \\
\partial_\gamma(\alpha + \beta) &= \partial_\gamma(\alpha) + \partial_\gamma(\beta) \\
\partial_\gamma(\alpha \& \beta) &= \partial_\gamma(\alpha) \,\&\, \partial_\gamma(\beta) \\
\partial_\gamma(\alpha^*) &= \partial_\gamma(\alpha) \sim \alpha^* \\
\partial_\gamma(\neg\alpha) &= \neg\partial_\gamma(\alpha)
\end{aligned}
\tag{2}
$$

What are regular expression derivatives?

$$
\begin{aligned}
\textbf{nullable}\,(S) &= \textbf{False} \\
\textbf{nullable}\,(\emptyset) &= \textbf{False} \\
\textbf{nullable}\,(\epsilon) &= \textbf{True} \\
\textbf{nullable}\,(\alpha \sim \beta) &= \textbf{nullable}\,(\alpha) \wedge \textbf{nullable}\,(\beta) \\
\textbf{nullable}\,(\alpha + \beta) &= \textbf{nullable}\,(\alpha) \vee \textbf{nullable}\,(\beta) \\
\textbf{nullable}\,(\alpha \& \beta) &= \textbf{nullable}\,(\alpha) \wedge \textbf{nullable}\,(\beta) \\
\textbf{nullable}\,(\alpha^*) &= \textbf{True} \\
\textbf{nullable}\,(\neg\alpha) &= \neg\textbf{nullable}\,(\alpha)
\end{aligned}
\tag{3}
$$

The big picture

- Each regular expression $\mathscr{L}$ represents a state in DFA.
- At each state $\mathscr{L}$, enumerate the alphabet Σ and create a edge from $\mathscr{L}$ to $\partial_\gamma(\mathscr{L})$ for character γ.
- Successful matches are represented by **nullable** expressions.
- Rejecting states are indicated by $\emptyset$.

The big picture

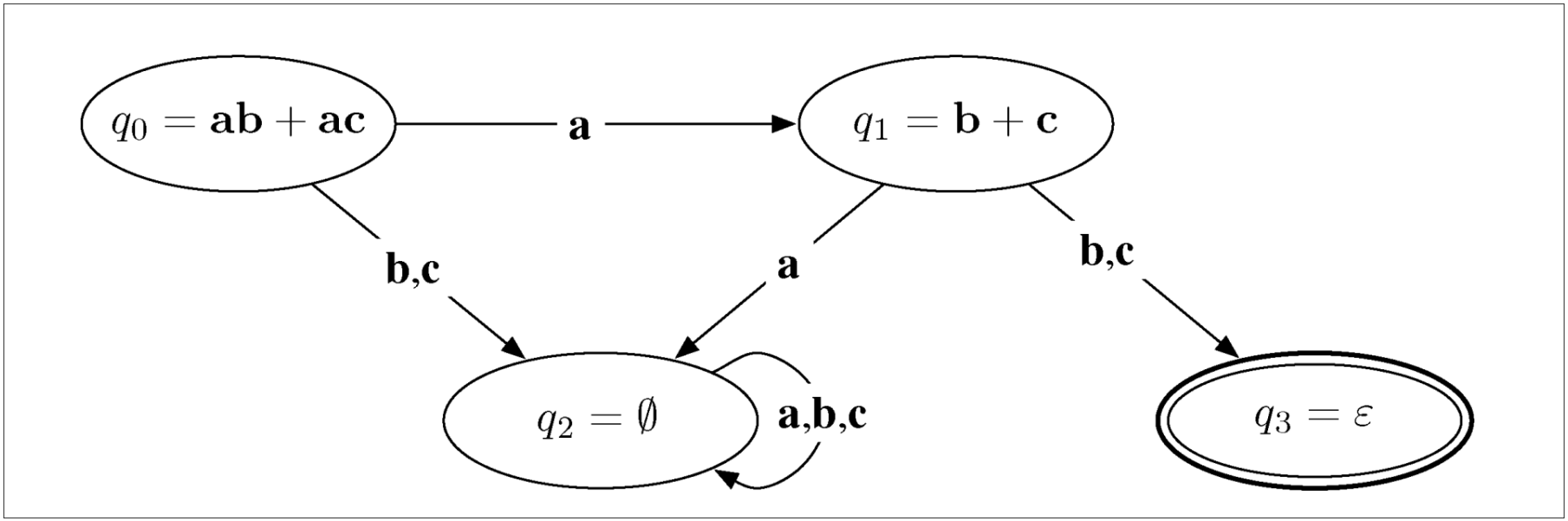

Figure: An example DFA for $(a \sim b) + (a \sim c)$

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

$$
\begin{array}{rcl}
r \,\&\, r & \approx & r \\
r \,\&\, s & \approx & s \,\&\, r \\
(r \,\&\, s) \,\&\, t & \approx & r \,\&\, (s \,\&\, t) \\
\emptyset \,\&\, r & \approx & \emptyset \\
\neg\emptyset \,\&\, r & \approx & r
\end{array}
\qquad
\begin{array}{rcl}
r \cdot s) \cdot t & \approx & r \cdot (s \cdot t) \\
\emptyset \cdot r & \approx & \emptyset \\
r \cdot \emptyset & \approx & \emptyset \\
\varepsilon \cdot r & \approx & r \\
r \cdot \varepsilon & \approx & r
\end{array}
$$

$$
\begin{array}{rcl}
r + r & \approx & r \\
r + s & \approx & s + r \\
(r + s) + t & \approx & r + (s + t) \\
\neg\emptyset + r & \approx & \neg\emptyset \\
\emptyset + r & \approx & r
\end{array}
\qquad
\begin{array}{rcl}
(r^*)^* & \approx & r^* \\
\varepsilon^* & \approx & \varepsilon \\
\emptyset^* & \approx & \varepsilon \\
\neg(\neg r) & \approx & r
\end{array}
$$

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 **Wnorm**($\mathscr{L}$) to represent apply weakly normalization on expression $\mathscr{L}$ based on this weaker notion of equivalence.

Congruence classes

Iterate through the whole alphabet?
Too costly for larger alphabets!

Congruence classes

Given two characters $\alpha, \beta \in \Sigma$ and a regular expression $\mathscr{L}$ (denoted by $\alpha \simeq_{\mathscr{L}} \beta$), we say α is **congruent** to β at expression $\mathscr{L}$ if $\partial_\alpha(\mathscr{L})$ and $\partial_\beta(\mathscr{L})$ accept the same set of languages.

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

$$d \simeq_{(a|b)\sim c} e \iff \partial_d((a|b) \sim c) \equiv \partial_e((a|b) \sim c) \equiv \emptyset$$

Congruence classes

- If we know all the congruence classes for $\mathscr{L}$, we only need to calculate the derivatives per class.
- Still no precise solution within reasonable time complexity.
- Finer partitions are safe!

Congruence classes

- If we know all the congruence classes for $\mathscr{L}$, we only need to calculate the derivatives per class.
- Still no precise solution within reasonable time complexity.
- Finer partitions are safe!

Congruence classes

$$
\begin{aligned}
\textbf{ApproxCC}(\epsilon) &= \{\Sigma\} \\
\textbf{ApproxCC}(\mathscr{S}) &= \{\mathscr{S}, \Sigma - \mathscr{S}\} \\
\textbf{ApproxCC}(\alpha \sim \beta) &= \begin{cases} \textbf{ApproxCC}(\alpha), & \neg\textbf{nullable}(\alpha) \\ \textbf{ApproxCC}(\alpha) \wedge \textbf{ApproxCC}(\beta), & \textbf{nullable}(\alpha) \end{cases} \\
\textbf{ApproxCC}(\alpha + \beta) &= \textbf{ApproxCC}(\alpha) \wedge \textbf{ApproxCC}(\beta) \\
\textbf{ApproxCC}(\alpha \& \beta) &= \textbf{ApproxCC}(\alpha) \wedge \textbf{ApproxCC}(\beta) \\
\textbf{ApproxCC}(\alpha^*) &= \textbf{ApproxCC}(\alpha) \\
\textbf{ApproxCC}(\neg\alpha) &= \textbf{ApproxCC}(\alpha) \\
S \wedge T &= \{s \cap t \mid s \in S \vee t \in T\}
\end{aligned}
\tag{4}
$$

Congruence classes

$$
\begin{aligned}
&\mathbf{ApproxCC}((a + (b \sim a)) \sim c) \\
=&\mathbf{ApproxCC}(a + (b \sim a)) \wedge \mathbf{ApproxCC}(c) \\
=&\mathbf{ApproxCC}(a) \wedge \mathbf{ApproxCC}(b \sim a) \wedge \mathbf{ApproxCC}(c) \\
=&\mathbf{ApproxCC}(a) \wedge \mathbf{ApproxCC}(b) \wedge \mathbf{ApproxCC}(c) \\
=&\{a, \Sigma - a\} \wedge \{b, \Sigma - b\} \wedge \{c, \Sigma - c\} \\
=&\{\emptyset, a, b, \Sigma - \{a, b\}\} \wedge \{c, \Sigma - c\} \\
=&\{\emptyset, a, b, c, \Sigma - \{a, b, c\}\}
\end{aligned}
$$

Precise result is $\{\{a, c\}, b, \Sigma - \{a, b, c\}\}$.

DFA generation

Algorithm 1 DFA states exploration

```
procedure ExploreDFA(x, V, T) ▷ x is current state (normalized), V is the set of visited states
    if x' ∈ V then
        return
    end if
    V ← V ∪ {x'}
    for c ∈ ApproxCC(x') do
        if IsEmpty(c) then
            continue
        end if
        r ← Representative(c)
        y ← Wnorm(∂_r(x'))
        T[x'][c] ← y                          ▷ update DFA transition table
        ExploreDFA(y, V, T)
    end for
end procedure
```

Figure: DFA States Exploration

Regular expression vectors for lexer generation

A **regular expression vector** $(\ell_1, \ell_2, \ldots, \ell_n) \in \mathscr{L}^n$ is just a tuple of regular expressions.

$$\begin{aligned}
\textbf{Wnorm}\left((\ell_1, \ldots, \ell_n)\right) &= (\textbf{Wnorm}(\ell_1), \ldots, \textbf{Wnorm}(\ell_n)) \\
\textbf{ApproxCC}\left((\ell_1, \ldots, \ell_n)\right) &= \bigwedge_{i=1}^{n} \textbf{ApproxCC}(\ell_i) \qquad (5) \\
\partial\left((\ell_1, \ldots, \ell_n)\right) &= (\partial(\ell_1), \ldots, \partial(\ell_n))
\end{aligned}$$

($\bigwedge$ is the meet operator defined in **ApproxCC**.)

Regular expression vectors for lexer generation

$$\begin{aligned} \textbf{Accept}\left((\ell_1, \ell_2, \ldots, \ell_n)\right) &= \bigvee_{i=1}^{n} \textbf{nullable}(\ell_i) \\ \textbf{Reject}\left((\ell_1, \ell_2, \ldots, \ell_n)\right) &= \bigwedge_{i=1}^{n} \ell_i \equiv \emptyset \end{aligned} \tag{6}$$

($\bigwedge, \bigvee$ are logical operators.)

Longest-match lexer

Algorithm 2 Longest-match Lexer

```
procedure LEXER(S)                                          ▷ S is the input
    M ← 0                                                   ▷ M is the longest match, initialized to 0 (no match)
    L ← 0                                                   ▷ L is offset of the longest match
    l ← LENGTH(S)
    s ← State_0
    for i ∈ RANGE(0, l) do
        c ← S[i]                                            ▷ c is set to EOF if the lexer is at the end of the input
        if s = State_0 then
            if comptime ACCEPT(State_0) then
                L ← i
                M ← comptime FIRSTACCEPT(State_0)
            end if
            if comptime REJECT(State_0) then
                break
            end if
            if c ∈ comptime APPROXCC(State_0)[0] then
                s ← comptime T[State_0][APPROXCC(State_0)[0]]
            end if
            go though all congruence classes
        end if
        go though all states
    end for
    return (M, L)
end procedure
```

Num of States

Lexer	`ml-lex`	`ml-ulex`	Minimal	Description
Burg	61	**58**	**58**	A tree-pattern match generator
CKit	122	**115**	**115**	ANSI C lexer
Calc	**12**	**12**	**12**	Simple calculator
CM	153	**146**	**146**	The SML/NJ compilation manager
Expression	**19**	**19**	**19**	A simple expression language
FIG	150	**144**	**144**	A foreign-interface generator
FOL	**41**	**41**	**41**	First-order logic
HTML	52	**49**	**49**	HTML 3.2
MDL	161	**158**	**158**	A machine-description language
ml-lex	121	**116**	**116**	The `ml-lex` lexer
Scheme	324	**194**	**194**	R^5RS Scheme
SML	251	**244**	**244**	Standard ML lexer
SML/NJ	169	**158**	**158**	SML/NJ lexer
Pascal	60	**55**	**55**	Pascal lexer
ml-yacc	100	**94**	**94**	The `ml-yacc` lexer
Russo	4803	3017	**2892**	System-log data mining
L_2	n/a	147	**106**	Monitoring stress-test

Preprocessing Time

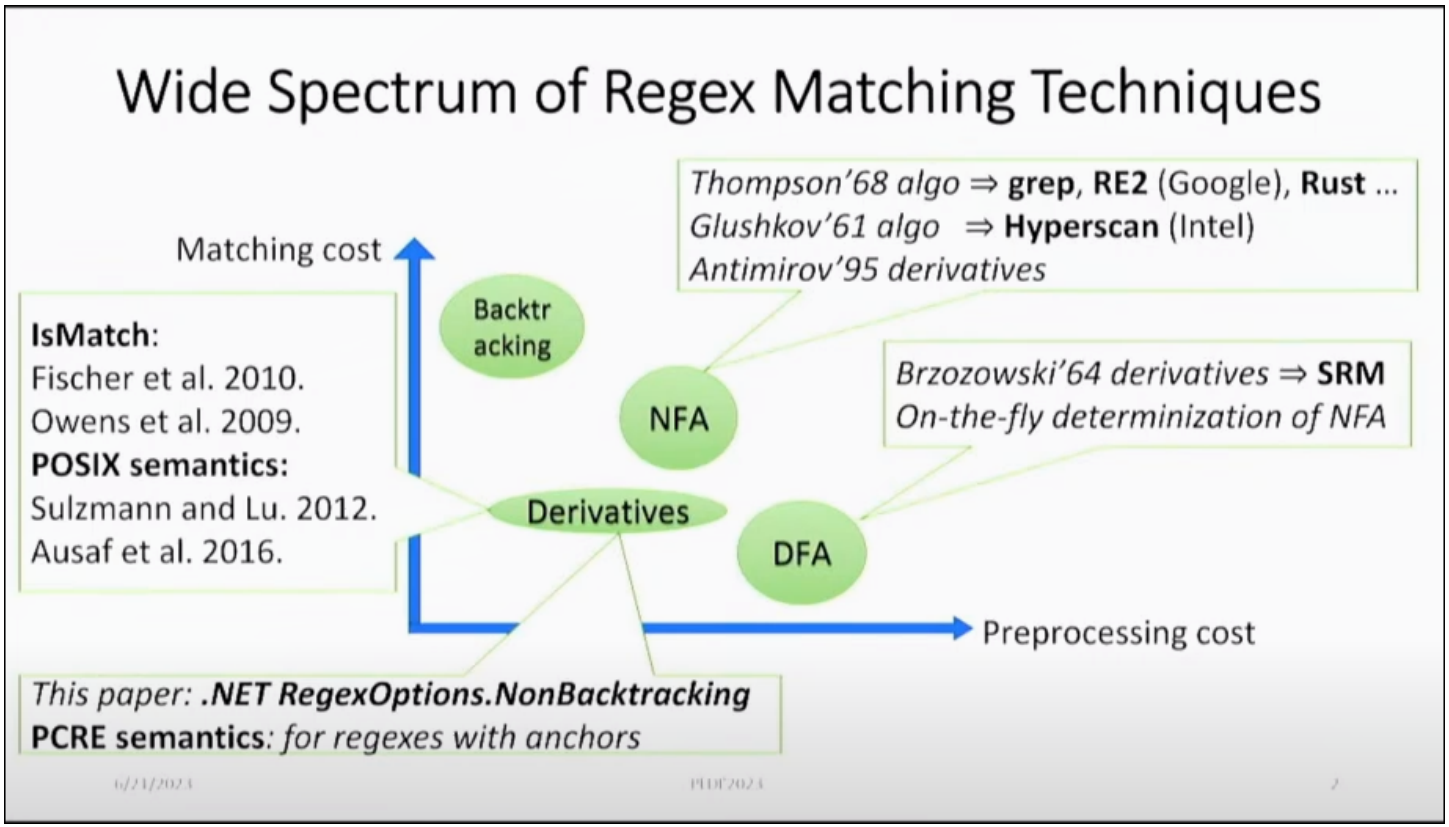

Figure: PLDI slides for Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics