Efficient Regular Expression Parsing

Fritz Henglein

Department of Computer Science,
University of Copenhagen

AiPL 2014, Edinburgh
August 23, 2014

Recall: Regular Expressions

- Regular Expressions (RE):

$$E ::= 0 \mid 1 \mid a \mid E_1 E_2 \mid E_1|E_2 \mid E_1^* \qquad (a \in \Sigma)$$

- Assume non-problematic REs: No REs containing sub-REs of the form E^* where E nullable.
  - All results extend to problematic REs, but are more complicated to state and prove.

What is Regular Expression “Matching”?

Given $s \in \Sigma^*$.

1. Acceptance testing: Is $s \in \mathcal{L}[\![E]\!]$?
   - String searching: Find some substring s' of s such that $s' \in \mathcal{L}[\![E]\!]$. (Variation: Find *all* substrings.)
2. Pattern matching: Given $s \in \Sigma^*$, find substrings of s such that each matches a *sub-RE* in E. (Variation: Return multiple matches for each sub-RE.)
3. Parsing: Return complete parse tree of s under E, if it exists

Note:

- Increasing information content.
- Classical automata theory (NFA->DFA, DFA minimization, etc.) applies only to acceptance testing.
- Pattern matching returns only one element match under $*$.

Example

RE = $((a|b)(c|d))^*$. Input string = *acbd*.

1. Acceptance testing: Yes!
2. Pattern matching: $(0, 4), (2, 4), (2, 3), (3, 4)$
3. Parsing: $[(\text{inl } a, \text{inl } c), (\text{inr } b, \text{inr } d)]$

Regular Expressions as Types

- Type interpretation $\mathcal{T}[\![E]\!]$:

$$
\begin{array}{rcl}
\mathcal{T}[\![0]\!] & = & \emptyset \\
\mathcal{T}[\![1]\!] & = & \{()\} \\
\mathcal{T}[\![a]\!] & = & \{a\} \\
\mathcal{T}[\![E_1\, E_2]\!] & = & \{(V_1, V_2) \mid V_1 \in \mathcal{T}[\![E_1]\!], V_2 \in \mathcal{T}[\![E_2]\!]\} \\
\mathcal{T}[\![E_1 | E_2]\!] & = & \{\mathsf{inl}\ V_1 \mid V_1 \in \mathcal{T}[\![E_1]\!]\} \\
 & & \cup \{\mathsf{inr}\ V_2 \mid V_2 \in \mathcal{T}[\![E_2]\!]\} \\
\mathcal{T}[\![E^*]\!] & = & \{[V_1, \ldots, V_n] \mid n \geqslant 0 \wedge \\
 & & \qquad \forall 1 \leqslant i \leqslant n.\, V_i \in \mathcal{T}[\![E]\!]\}
\end{array}
$$

- Value = parse tree = proof of inhabitation

Unparsing ("Flattening")

- Flattening yields underlying string:

$$
\begin{aligned}
\mathsf{flat}(()) &= \epsilon \\
\mathsf{flat}(a) &= a \\
\mathsf{flat}((V_1, V_2)) &= \mathsf{flat}(V_1)\,\mathsf{flat}(V_2) \\
\mathsf{flat}(\mathsf{inl}\ V_1) &= \mathsf{flat}(V_1) \\
\mathsf{flat}(\mathsf{inr}\ V_2) &= \mathsf{flat}(V_2) \\
\mathsf{flat}([V_1, \ldots, V_n]) &= \mathsf{flat}(V_1) \cdots \mathsf{flat}(V_n)
\end{aligned}
$$

- The parse trees for a given string s:
  $\mathcal{T}_s[\![E]\!] = \{V \in \mathcal{T}[\![E]\!] \mid \mathsf{flat}(V) = s\}$.

Proposition

$\mathcal{L}[\![E]\!] = \{\mathsf{flat}(V) \mid V \in \mathcal{T}[\![E]\!]\}$.

Challenges

- Grammatical ambiguity: Which parse tree to return?
- How to represent parse trees compactly?
- Time: Straightforward backtracking algorithm, but impractical: $\Theta(m 2^n)$ time, where $m = |E|$, $n = |s|$.
- Space: How to minimize RAM consumption?

Disambiguation

- RE E ambiguous iff $|\mathcal{T}_s[\![E]\!]| > 1$ for some s.
- How to deterministically choose one $V \in \mathcal{T}_s[\![E]\!]$ among several possible candidates?
- Greedy matching: Intuitively, choose what a backtracking parser returns:
  1. Try left alternative of $E|F$ first.
  2. If it fails, backtrack and try the right alternative.
  3. Treat E^* as $EE^*|1$.

Greedy Order $\prec_{\mathcal{V}}$

$$
\begin{array}{rcll}
\text{inl } V & \prec_{\mathcal{V}} & \text{inr } V' & \\
[V_1, \ldots] & \prec_{\mathcal{V}} & [] & \\
(V_1, V_2) & \prec_{\mathcal{V}} & (V_1', V_2') & \text{if} \quad V_1 \prec_{\mathcal{V}} V_2 \vee \\
 & & & \qquad (V_1 = V_1' \wedge V_2 \prec_{\mathcal{V}} V_2') \\
\text{inl } V & \prec_{\mathcal{V}} & \text{inl } V' & \text{if} \quad V \prec_{\mathcal{V}} V' \\
\text{inr } V & \prec_{\mathcal{V}} & \text{inr } V' & \text{if} \quad V \prec_{\mathcal{V}} V' \\
[V_1, \ldots] & \prec_{\mathcal{V}} & [V_1', \ldots] & \text{if} \quad V_1 \prec_{\mathcal{V}} V_1' \\
[V_1, V_2, \ldots] & \prec_{\mathcal{V}} & [V_1, V_2', \ldots] & \text{if} \quad [V_2, \ldots] \prec_{\mathcal{V}} [V_2', \ldots]
\end{array}
$$

Proposition (Frisch/Cardelli)

For any nonproblematic RE E, string s, $\prec_{\mathcal{V}}$ is a strict well-founded total order on $\mathcal{T}_s[\![E]\!]$.

Definition

Greedy parse for $s \in \mathcal{L}[\![E]\!]$: $\min_{\prec_{\mathcal{V}}} \mathcal{T}_s[\![E]\!]$.

Bit-Coding

- Compact representation of parse trees where the RE is known.
- Encoding $\ulcorner \cdot \urcorner : \mathcal{V} \to \{1, 0\}^*$,

$$
\begin{array}{rcl}
\ulcorner () \urcorner & = & \epsilon \\
\ulcorner a \urcorner & = & \epsilon \\
\ulcorner (V_1, V_2) \urcorner & = & \ulcorner V_1 \urcorner \ulcorner V_2 \urcorner \\
\ulcorner \mathrm{inl}\ (V_1) \urcorner & = & 0 \ulcorner V_1 \urcorner \\
\ulcorner \mathrm{inr}\ (V_2) \urcorner & = & 1 \ulcorner V_2 \urcorner \\
\ulcorner [V_1, \ldots, V_n] \urcorner & = & 0 \ulcorner V_1 \urcorner \cdots 0 \ulcorner V_n \urcorner\, 1
\end{array}
$$

- Type-indexed decoding $\llcorner \cdot \lrcorner_E : \{1, 0\}^* \rightharpoonup \mathcal{T}[\![E]\!]$: Interpret RE as nondeterministic algorithm to construct parse tree, with bit-code as oracle. (“Every bit counts.”)
- $\mathcal{B}[\![E]\!] = \{\ulcorner V \urcorner \mid V \in \mathcal{T}[\![E]\!]\}$
  - $\mathcal{B}_s[\![E]\!] = \{\ulcorner V \urcorner \mid V \in \mathcal{T}_s[\![E]\!]\}$.

Example

RE = $((a|b)(c|d))^*$. Input string = *acbd*.

1. Acceptance testing: Yes!
2. Pattern matching: $(0, 4), (2, 4), (2, 3), (3, 4)$
3. Parsing: $[(\text{inl } a, \text{inl } c), (\text{inr } b, \text{inr } d)]$
    - Bit-code: 0 00 0 11 1.

Bit-coding: Examples

- Bit codes for the string `abcbcba`

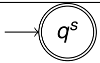
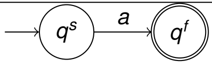
Regular expression	Representation	Size
Latin1	abcbcba00000000	64
Σ^*	0a0b0c0b0c0b0a1	64
$((a+b)+(c+d))^*$	0000010100010100010001	22
$a \times b \times c \times b \times c \times b \times a$		0

Augmented NFAs

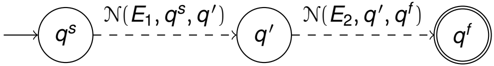
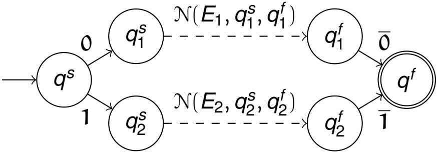
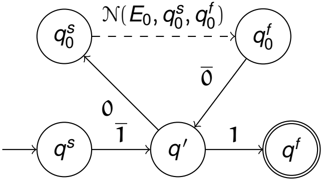
- Augmented NFA (aNFA) is a 5-tuple $M \in (Q, \Sigma, \Delta, q^s, q^f)$.
- States Q; $q^s, q^f \in Q$ start and finishing states.
- Input alphabet Σ.
- Labeled transition relation $\Delta \subseteq Q \times (\Sigma \cup \{1, 0\} \cup \{\bar{1}, \bar{0}\}) \times Q$.
  - Σ *input labels*; $\{1, 0\}$ *output labels*; $\{\bar{1}, \bar{0}\}$ *log labels*.
- Write $q \overset{p}{\rightsquigarrow} q'$ if there is a walk from q to q'; p sequence of labels.
  - $\mathrm{in}(p)$ = input label subsequence;
  - $\mathrm{out}(p)$ = output labels;
  - $\log(p)$ = log labels.

aNFA Construction (1/2)

- Define $\mathcal{N}(E, q^s, q^f)$ as set of aNFAs for E, with start and finishing states q^s, q^f:

E	$\mathcal{N}(E, q^s, q^f)$
0	
1	 (implies $q^s = q^f$)
a	

aNFA Construction (2/2)

E	$\mathcal{N}(E, q^s, q^f)$
$E_1 \times E_2$	
$E_1 + E_2$	
E_0^*	

Representation Theorem

Theorem

Let $M = \mathcal{N}(E, q^s, q^f)$. The paths of M are in one-to-one correspondence with the parse trees of E:

$$\mathcal{B}_s[\![E]\!] = \{\mathrm{out}(p) \mid q^s \overset{p}{\rightsquigarrow} q^f, \mathrm{in}(p) = s\}$$

- Important to use Thompson-style ϵ-NFAs! Does not hold for DFAs, ϵ-free NFAs.
- Already observed by Brüggemann-Klein (1993).

Greedy parse = Lexicographically least bitcode

Proposition

For all E, $V, V' \in \mathcal{T}[\![E]\!]$:

$$V \prec_{\mathcal{V}} V' \iff \ulcorner V \urcorner \prec_{\mathcal{B}} \ulcorner V' \urcorner$$

where $\prec_{\mathcal{B}}$ is lexicographic ordering on $\{0,1\}^*$.

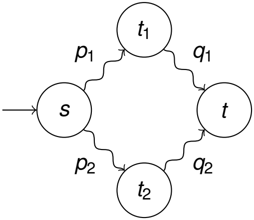
Corollary

Let $M = \mathcal{N}(E, q^s, q^f)$. For all $s \in \mathcal{L}[\![E]\!]$:

$$\min_{\prec_{\mathcal{V}}} \mathcal{T}_s[\![E]\!] = \llcorner \min_{\prec_{\mathcal{B}}} \{\mathrm{out}(p) \mid q^s \overset{p}{\rightsquigarrow} q^f, \mathrm{in}(p) = s\} \lrcorner_E.$$

Monotonicity of $\prec_{\mathcal{B}}$

Proposition



If p_1 not prefix of p_2, then

$$\mathrm{out}(p_1) \prec_{\mathcal{B}} \mathrm{out}(p_2) \Rightarrow \mathrm{out}(p_1 q_1) \prec_{\mathcal{B}} \mathrm{out}(p_2 q_2)$$

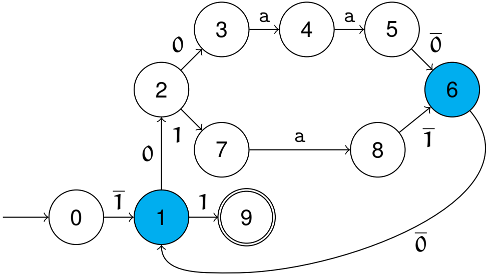
Lean-log algorithm

- Simulate aNFA for input *s*, using ordered state sets.
  - Each state represents *lexicographically least path* reading current prefix.
  - States are ordered according to the lexicographic ordering on the paths they represent.
- Perform state-ordered ϵ-closure: Log 1 bit per join state for each input character.
- After complete string is processed, use reverse aNFA and log bits to construct lexicographically least bit-code.
- (Construct parse tree from bit-code, if desired.)

Example: Parse aaa with RE $(aa|a)^*$

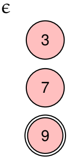
- Input: aaa

Log	ϵ	a	a	a
1:				
6:				

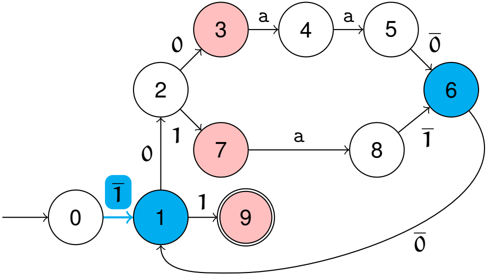


Example: Parse aaa with RE $(\mathtt{aa}|\mathtt{a})^*$

- Input: aaa

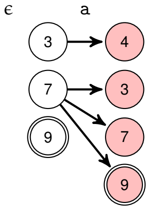


Log	ϵ	a	a	a
1:	$\overline{1}$			
6:	-			

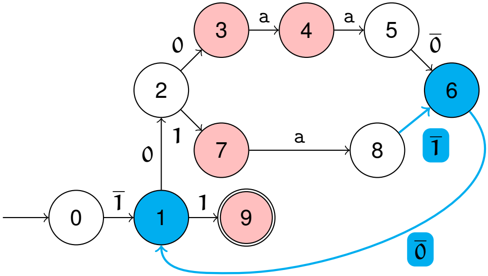


Example: Parse aaa with RE $(\texttt{aa}|\texttt{a})^*$

- Input: a aa

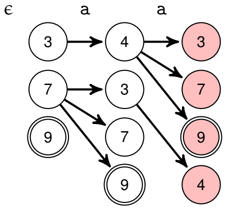


Log	ϵ	a	a	a
1:	$\bar{1}$	$\bar{0}$		
6:	-	$\bar{1}$		

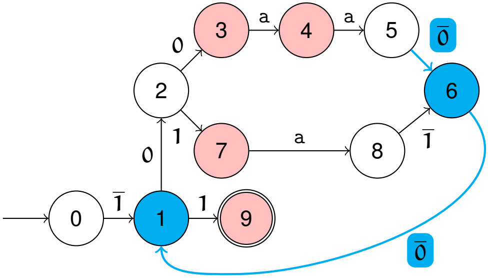


Example: Parse aaa with RE (aa|a)*

- Input: aa a

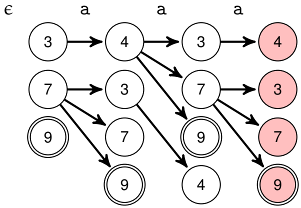


Log	ϵ	a	a	a
1:	$\bar{1}$	$\bar{0}$	$\bar{0}$	
6:	-	$\bar{1}$	$\bar{0}$	

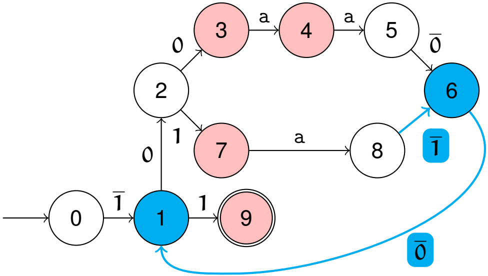


Example: Parse aaa with RE $(aa|a)^*$

- Input: aaa



Log	ϵ	a	a	a
1:	$\bar{1}$	$\bar{0}$	$\bar{0}$	$\bar{0}$
6:	-	$\bar{1}$	$\bar{0}$	$\bar{1}$



Key properties of lean-log algorithm

- Semi-streaming: Forward streaming pass over input, logging join-state bits; backward pass for constructing bit-code.
  - Two passes because of disambiguation requiring unbounded look-ahead.
- Input string read in streaming fashion, using $O(m)$ working memory and kn bits of LIFO memory for the log, $k =$ number of alternatives and stars in E.
- Input string need not be stored. (Consider input coming from a generator.)
- Runs in time $O(mn)$.

Implementation

- Implementations of lean-log algorithm
  - Straightforward Haskell version
  - Optimized Haskell version, based on Conduit (10 times faster and)
  - Straightforward C version (10 times faster than fast Haskell version)
- No NFA-minimization, no DFA generation, no word-level parallelism, no special RE-processing, no special handling of bounded iteration.

Performance

- Better performance than Play
- Competitive with RE2 when RE2 does not employ static optimizations, or when subjected to REs that are not “tuned” to Perl (made deterministic)
- Otherwise competitive with Grep and other tools, but not with RE2.
  - These tools perform only acceptance testing or RE pattern matching, not full parsing; and they don’t always do it correctly.
- Best amongst all tested full RE parsers (both greedy and other).

Regular matching algorithms

Problem	Time	Space	Aux	Answer
NFA simulation	$O(mn)$	$O(m)$	0	0/1
Perl	$O(m2^n)$	$O(m)$	0	k groups
RE2[1]	$O(mn)$	$O(m+n)$	0	k groups
Parse (3-p)[2]	$O(mn)$	$O(m)$	$O(n)$	greedy parse
Parse (2-p)[3]	$O(mn)$	$O(m)$	$O(n)$	greedy parse
Parse (str.)[4]	$O(mn+f(m))$	$O(m)$	$O(n)$	greedy parse

(n size of input, m size of RE)

[1] Cox (2007)

[2] Frisch, Cardelli (2004)

[3] Grathwohl, Henglein, Nielsen, Rasmussen (2013)

[4] Grathwohl, Henglein, Rasmussen (2014)

Summary

- Regular expression *parsing*: Return *list* of matches for Kleene star, not just last match.
- Greedy parse = least parse in Greedy order. (Correspondingly for POSIX)
- Greedy-ordered parse = what backtracking yields.
- Greedy parse can be computed without *any* backtracking.
  - NFA-simulation with *ordered* state sets.
- Worst-case *linear time* parsing for fixed RE (= scalable, guaranteed no REDoS)
- Bit-coding $\cong$ parse tree minus underlying RE
- Semi-streaming and optimally streaming algorithms
  - Input need not be stored in memory before, during or after parsing.
  - RAM requirements independent of size of input string.

End