Kleenex: From nondeterministic finite state transducers to streaming string transducers

Fritz Henglein

DIKU, University of Copenhagen

2015-05-28
WG 2.8 meeting, Kefalonia
Joint work with Bjørn Bugge Grathwohl, Ulrik Terp Rasmussen, Kristoffer Aalund Søholm and Sebastian Paaske Tørholm (DIKU)
Streaming regular expression processing

Input:
- Regular expression (maybe annotated)
- Stream of characters

Output:
- Parse tree
- Parse tree, but with parts left out (includes subgroup matching)
- Parse tree, but with parts substituted

Examples:
- Web-UI data (issuu.com, JSON, 10 TB/month)
- DNA (UCPH Department of Biology, text, 1 PB stored)
- High-frequency trading (X, Y, continuous)

Think Perl regex processing.
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? How to stream?
Regular Expressions as Types

- Regular Expressions (RE):

\[ E ::= 0 | 1 | a | E_1 E_2 | E_1 | E_2 | E_1^* \quad (a \in \Sigma) \]

- Type interpretation \( \mathcal{T}[E] \):

\[
\begin{align*}
\mathcal{T}[0] & = 0 = \emptyset \\
\mathcal{T}[1] & = 1 = \{(())\} \\
\mathcal{T}[a] & = \{a\} = \{a\} \\
\mathcal{T}[E_1 E_2] & = E_1 \times E_2 = \{(V_1, V_2) \mid V_1 \in \mathcal{T}[E_1], V_2 \in \mathcal{T}[E_2]\} \\
\mathcal{T}[E_1 | E_2] & = E_1 + E_2 = \{\text{inl } V_1 \mid V_1 \in \mathcal{T}[E_1]\} \cup \{\text{inr } V_2 \mid V_2 \in \mathcal{T}[E_2]\} \\
\mathcal{T}[E^*] & = E \text{ list} = \{[V_1, \ldots, V_n] \mid n \geq 0 \land \forall 1 \leq i \leq n. V_i \in \mathcal{T}[E]\}
\end{align*}
\]

- Not the language interpretation \( \mathcal{L}[E] \)!

- “Value” = Element of type = parse tree = proof of inhabitation

Bit-Coding: Serialized parse trees

- Prefix code for parse trees.
- Encoding \( \cdot \downarrow : \mathcal{V} \to \{1, 0\}^* \),

\[
\begin{align*}
\begin{array}{ll}
\downarrow (\ ) & = \epsilon \\
\downarrow a & = \epsilon \\
\downarrow (V_1, V_2) & = \downarrow V_1 \downarrow V_2 \\
\downarrow \text{inl} (V_1) & = 0\downarrow V_1 \\
\downarrow \text{inr} (V_2) & = 1\downarrow V_2 \\
\downarrow [V_1, \ldots, V_n] & = 0\downarrow V_1 \cdots 0\downarrow V_n 1
\end{array}
\end{align*}
\]

- **Type-indexed decoding** \( \downarrow \downarrow_E : \{1, 0\}^* \to \mathcal{T}[E] \): Interpret RE as nondeterministic algorithm to construct parse tree, with bit-code as oracle.

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: [(inl a, inl c), (inr b, inr d)]
   ▶ Bit-code: 0 0 0 1 1 1.
## Bit-coding: Examples

- Bit codes for the string `abcbcba`

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Representation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latin1</td>
<td><code>abcbcba00000000</code></td>
<td>64</td>
</tr>
<tr>
<td>Σ*</td>
<td><code>0a0b0c0b0c0b0a1</code></td>
<td>64</td>
</tr>
<tr>
<td><code>((a + b) + (c + d))</code>*</td>
<td><code>0000010100010100010001</code></td>
<td>22</td>
</tr>
<tr>
<td><code>a × b × c × b × c × b × a</code></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Augmented Thompson NFAs

- Thompson NFA with output labels on split- and join-nodes.
- Construction:

<table>
<thead>
<tr>
<th>$E$</th>
<th>$\mathcal{N}(E, q^s, q^f)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$q^s \rightarrow q^f$</td>
</tr>
<tr>
<td>1</td>
<td>$q^s \rightarrow q^f$ (implies $q^s = q^f$)</td>
</tr>
<tr>
<td>$a$</td>
<td>$q^s \rightarrow q^f$</td>
</tr>
</tbody>
</table>
Augmented Thompson NFAs

\[ E \rightarrow \mathcal{N}(E, q^s, q^f) \]

\[ E_1 E_2 \rightarrow \mathcal{N}(E_1, q^s, q') \times q' \rightarrow \mathcal{N}(E_2, q', q^f) \rightarrow q^f \]

\[ E_1 | E_2 \rightarrow \mathcal{N}(E_1, q^s_1, q^f_1) \]

\[ E_0^* \rightarrow \mathcal{N}(E_0, q^s_0, q^f_0) \]

Simplification: \( \overline{0} \)- and \( \overline{1} \)-labeled edges contracted.
Augmented Thompson NFA: Example

Augmented Thompson NFA for $a^* b | (a | b)^*$
Representation Theorem

**Theorem**

*One-to-one correspondence between*

- *parse trees for E*,
- *paths in augmented Thompson automaton for E*,
- *bit-coded parse trees = bit subsequences of automaton paths.*

*Lexicographically least bit-code = greedy parse.*

- Important to use Thompson-style $\epsilon$-NFAs. Does not hold for DFAs, $\epsilon$-free NFAs.
Optimal streaming

- Assume partial $f : \Sigma^* \leftrightarrow \Delta^*$.  
  - Example: Bit-coded greedy parse of input sequence
- Optimally streaming version of $f$:
  
  $$f^#(s) = \prod\{f(ss') \mid ss' \in \text{dom} f\}$$

  where $\prod$ = longest common prefix.

- Outputs bits as soon as those are semantically determined by the prefix seen so far.
### Regular matching algorithms

<table>
<thead>
<tr>
<th>Problem</th>
<th>Time</th>
<th>Space</th>
<th>Aux</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA simulation</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>0</td>
<td>0/1</td>
</tr>
<tr>
<td>Perl</td>
<td>$O(m2^n)$</td>
<td>$O(m)$</td>
<td>0</td>
<td>$k$ groups</td>
</tr>
<tr>
<td>RE2$^1$</td>
<td>$O(mn)$</td>
<td>$O(m + n)$</td>
<td>0</td>
<td>$k$ groups</td>
</tr>
<tr>
<td>Parse (3-p)$^2$</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
<tr>
<td>Parse (2-p)$^3$</td>
<td>$O(mn)$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
<tr>
<td>Parse (str.)$^4$</td>
<td>$O(mn + 2^{m\log m})$</td>
<td>$O(m)$</td>
<td>$O(n)$</td>
<td>greedy parse</td>
</tr>
</tbody>
</table>

($n$ size of input, $m$ size of RE)

---

$^1$ Cox (2007)  
$^2$ Frisch, Cardelli (2004)  
$^3$ Grathwohl, Henglein, Nielsen, Rasmussen (2013)  
$^4$ Optimally streaming. Grathwohl, Henglein, Rasmussen (2014)
Augmented Thompson NFA: Example

Augmented Thompson NFA for $a^* b | (a | b)^*$
Augmented Thompson NFA as NFST

Augmented Thompson NFA for $a^*b|(a|b)^*$
Generalizations

Techniques work for *arbitrary* NFSTs:
- arbitrary outputs (and output actions), not just $\epsilon$ and individual bits;
- intuitively fusion of parsing with subsequent catamorphism.

NFSTs (with $\epsilon$-transitions) are more compact than RE.
- DFA as RE: $\Omega(m^2)$ blow-up.
- NFA as $\epsilon$-free NFA (matrix representation): $\Omega(m \log m)$ blow-up;
  standard construction (Glushkov): $\Theta(m^2)$ blow-up.
- NFSTs correspond to left-linear grammars with output actions.
- Kleenex: Surface language for linear grammars with output actions.
Determinization: Streaming string transformers

- **Streaming string transducer:**
  - deterministic finite automata,
  - each state equipped with fixed number of registers containing strings
  - registers updated on transition by affine function;

- **Determinization:**
  - Finite number of possible path trees during NFST-simulation
  - Edges in a path tree $\cong$ registers
Determinization: Example

\[ x_0 := (x_0)(x_{00}) \]
\[ x_1 := (x_1)(x_{10})(x_{100}) \]
\[ x_{00}, x_{100}, x_{10} := 0 \]
\[ x_{01}, x_{101}, x_{11} := 1 \]

\[ x_0, x_{00}, x_{10}, x_{100} := 0 \]
\[ x_{01}, x_1, x_{11}, x_{101} := 1 \]

\[ S_{5,9,7,8,4} \]

\[ x_0 := (x_0)(x_{01}) \]
\[ x_1 := (x_1)(x_{10})(x_{101})0 \]
\[ x_{10} := 0 \]
\[ x_{11} := 1 \]

\[ x_0 := (x_0)(x_{01}) \]
\[ x_1 := (x_1)(x_{10})(x_{101})0 \]
\[ x_{10} := 0 \]
\[ x_{11} := 1 \]

\[ S_{4,7,8} \]

\[ x_e := (x_e)(x_1)(x_{10}) \]
\[ a/ x_0, x_{00} := 0 \]
\[ x_1, x_{01} := 1 \]

\[ x_e := (x_e)(x_0)(x_{00}) \]
\[ a/ x_0, x_{00} := 0 \]
\[ x_1, x_{01} := 1 \]

\[ S_{7,8,4} \]

\[ x_e := (x_e)(x_1)(x_{11}) \]
\[ b/ x_0, x_{00} := 0 \]
\[ x_1, x_{01} := 1 \]

\[ S_{4,7,8} \]

\[ x_e := (x_e)(x_1)(x_{11}) \]
\[ b/ x_0, x_{00} := 0 \]
\[ x_1, x_{01} := 1 \]

\[ x_e := (x_e)(x_0)(x_{01}) \]
\[ b/ x_0, x_{00} := 0 \]
\[ x_1, x_{01} := 1 \]

\[ S_{7,8,4} \]
Implementation

- Compilation of Kleenex to streaming string transformer in Haskell;
- generates C code (goto-form), linked with string concatenation library.
- Optimizations: Lookahead processing, symbolic transitions, register constant propagation.
Performance evaluation

- Comparison RE2, RE2J, Oniglib, Ragel, awk, sed, grep, Perl, Python, specialized tools.
- Standard desktop
- Single-core Kleenex:
  - High throughput even for complex specifications
  - Typically around 1 Gb/s, for simple specifications more (6 Gb/s)
Performance test: Issuu simple

{{("[a-z_]*":(-?[0-9]*|"(([^"]|\")*)")?,?)\n?)*}}
Performance test: Issuu

({"((((visitor_username|(visitor_uuid|visitor_source))|(visitor_useragent|visitor_referrer) |(visitor_country|visitor_device))) |(((visitor_ip|env_type)|(env_doc_id|env_adid)) |((env_ranking|env_build)|(env_name|env_component)))) |(((event_type|event_service)|(event_readtime |event_index)))|((subject_type|subject_doc_id) |(subject_page|subject_infoboxid)))|((subject_url |subject_link_position)|(cause_type|cause_position)) |((cause_adid|cause_embedid)|(cause_token|cause)))" :(?\[0-9]*"("((internal|external)|([A-Z][A-Z])|(browser |android))))|([0-9a-f]{16}|reader)|(stream|(website |impression)))|((click|read)|(download|(share |pageread)))|((pagereadtime|(continuation_load|doc)) |(infobox|(link|page)))|(((ad|related)|(archive |embed|email)))|((facebook|(twitter|google))|(tumblr |linkedin|0-9]{12}-[a-z0-9]{32})))|((Mozilla/ |Windows NT)|(WOW64|Linux|Android)))|((Mobile |(WebKit/|(KHTML, like Gecko)))|((Chrome/|(Safari/ |(["\[\"]\"\")*))))),?)*\n?)*
Towards 5 Gbps/core

- Multistriding with tabling (8 bytes at a time)
- Transducer optimizations (shrinking)
- Hardware- and systems-specific optimizations
Future work

- Parallel RE processing
  - Mytkowicz et al. (ASPLOS 2014, PPoPP 2014, POPL 2015)
- Optimally streaming substitution and aggregation
- Probabilistic matching
- ...
- Characterization of 1NFSTs
- Visibly PDAs/nested word automata
- ...
- Applications (bioinformatics, finance, weblogs, ... )
Summary

- Regular expressions as types
  - Grammars as types
- Bitcoding
- Augmented Thompson NFAs
- Characterization: (lex. least) path = (greedy) parse tree
- Optimal streaming
- (Augmented Thompson NFA simulation)
- Determinization: Streaming string transformers
- ...to get raw speed.

More information: www.diku.dk/kmc.