Martin Berglund, Willem Bester & Brink van der Merwe

Formalising Boost POSIX Regular Expression Matching

15th International Colloquium on Theoretical Aspects of Computing

18 October 2018, Stellenbosch, South Africa

UNIVERSITEIT iYUNIVESITHI STELLENBOSCH UNIVERSITY

100

1918 · 2018

What we've been doing

We've been thinking about

- regular expression matching semantics
  - Perl-Compatible Regular Expression (PCRE) engines
  - POSIX-compliant engines
- ambiguity — "more than one way to match"
- capture groups

Why Boost?

- "very powerful" C++ library
- mature (1999–)
- online peer-reviewed QA process
- regular expression engine that has a POSIX mode

Leftmost-greedy vs leftmost-longest matching

Match "aba"
with $E_1 = \underline{\texttt{(ab|ba|a)*}}$

ambiguous

[ab][a] [a][ba]

Leftmost-greedy	[ab][a]
Leftmost-longest	[ab][a]

Match "aba"
with $E_2 = \underline{\texttt{(a|ab|ba)*}}$

ambiguous

[ab][a] [a][ba]

Leftmost-greedy	[a][ba]
Leftmost-longest	[ab][a]

- E_2 defines the same language as E_1, but subexpression order differs
  - Compare $E_1 = \underline{\texttt{(ab|ba|a)*}}$ to $E_2 = \underline{\texttt{(a|ab|ba)*}}$
- Leftmost-longest: matcher *seemingly* considers all possible matches for subexpressions [more on this later]

The POSIX regular expression specification

POSIX specifies leftmost-longest matching:

"The search for a matching sequence starts at the beginning of a string and stops when the first sequence matching the expression is found, where 'first' is defined to mean 'begins earliest in the string'. If the pattern permits a variable number of matching characters and thus there is more than one such sequence starting at that point, the longest such sequence is matched.... Consistent with the whole match being the longest of the leftmost matches, each ***subpattern*** from left to right shall match the longest possible string."

Fowler's complaint: "Subpattern" only used here; elsewhere it's "subexpression" (always in the context of grouping).

Note: We only consider full matching in this work.

An eccentric reading of the POSIX standard?

Match "aba"
with `(ab|ba|a)*`

Regex-TDFA:	`[ab][a]`
Boost:	`[a][ba]`

Match "aba"
with `(a|ab|ba)*`

Regex-TDFA:	`[ab][a]`
Boost:	`[a][ba]`

Regex-TDFA written in Haskell.
Boost written in C++.

POSIX

- Full matching with submatch addressing
- Position and extent of substrings matched by subexpressions must be available

Boost POSIX Mode

- Maximises what is reported for marked subexpressions (those surrounded by parentheses)
- Essentially, reading POSIX with:
  `s/subpattern/marked subexpression/`

More examples

Match "aa"
with $(_0(_1a*)_1(_2a*)_2)_0$

Captures	Boost	RTDFA
$[_0[_1aa]_1[_2]_2]_0$	✓	✓
$[_0[_1a]_1[_2a]_2]_0$		
$[_0[_1]_1[_2aa]_2]_0$		

Match "aa"
with $(_0a*(_1a*)_1)_0$

Captures	Boost	RTDFA
$[_0aa[_1]_1]_0$		✓
$[_0a[_1a]_1]_0$		
$[_0[_1aa]_1]_0$	✓	

Note: All non-atomic subexpressions are parenthesised.

- Regex-TDFA maximises lengths of ***all subexpressions*** in the order they occur in the regular expression
- Boost maximises lengths of ***(capture) groups*** in the order they occur in the regular expression

Capturing regular expressions and forests

Capturing Regular Expressions

Over a finite alphabet Σ and an index set I:

$\varnothing$	empty language **ATOM**
ε	empty string **ATOM**
a	symbols $a \in \Sigma$ **ATOM**
$(r_0 \cdot r_1)$	concatenation of capturing regular expression r_0, r_1
$(r_0 + r_1)$	alternation of capturing regular expressions r_0, r_1
(r^*)	closure of capturing regular expression r
$({}_i r)_i$	capture group $i \in \Sigma$ of capturing regular expression r

Set of Forests

Over a finite alphabet Σ
and an index set I:

- $(\Sigma \cup \{\varepsilon\})$ is a forest
- So is $f_1 f_2$ for forests f_1 and f_2
- And $[{}_i f]_i$ for forest f and $i \in I$

Note

If I is non-empty: the strings over Σ properly contained in the set of forests.

If I is empty: they are equal.

Forest and String Languages

Forest Language

$\mathscr{L}(r)$ for a capturing regular expression r

$$\begin{aligned}
\mathscr{L}(\varnothing) &= \varnothing \\
\mathscr{L}(\varepsilon) &= \{\varepsilon\} \\
\mathscr{L}(a) &= \{\mathsf{a}\} \\
\mathscr{L}(r_0 \cdot r_1) &= \mathscr{L}(r_0) \cdot \mathscr{L}(r_1) \\
\mathscr{L}(r_0 + r_1) &= \mathscr{L}(r_0) \cup \mathscr{L}(r_1) \\
\mathscr{L}(r^*) &= \mathscr{L}(r)^* \\
\mathscr{L}((_i r)_i) &= \{[_i\} \cdot \mathscr{L}(r) \cdot \{]_i\}
\end{aligned}$$

- For string w over Σ, and $\Sigma' \subseteq \Sigma$: $\pi_{\Sigma'}(w)$ is the maximal subsequence of w that contains only symbols from Σ'.
- The string language described by the capturing regular expression r over Σ is the set $\pi_\Sigma(\mathscr{L}(r))$.

Also: By extension, we also handle

$$r? = (r + \varepsilon), \quad r^+ = rr^*, \quad \text{and} \quad r^{m,n} = \underbrace{r \cdots r}_{m \text{ times}} \underbrace{(r+\varepsilon) \cdots (r+\varepsilon)}_{n \text{ times}}$$

From forest to captures

Strategy to compute capture information

1. collect the matching forests
2. determine the capture history $C(f)$ and final capture history $C_{\text{fin}}(f)$ for each forest f
3. order forests by Boost partial order $\prec_B$ on C_{fin} values
4. return the greatest C_{fin} value as determined by $\prec_B$

Capture history

- informally, a function $C(f, i)$ for forest f and group i
- returns a pair (s, ℓ) for each substring captured by group i
- $s \leftarrow$ substring start index, $\ell \leftarrow$ substring length

Final capture history

- $C_{\text{last}}(f, i)$ is the pair (s, ℓ) in $C(f, i)$ with the greatest s
- $C_{\text{fin}}(f)$ is the set $\{(j, C_{\text{last}}(f, j) \mid j \in I\}$

Boost partial order and captures

Boost partial order

- denote as $\prec_B$
- assume $\pi_\Sigma(f_1) = \pi_\Sigma(f_2)$

Then $C_{\text{fin}}(f_1) \prec_B C_{\text{fin}}(f_2)$ if for the smallest $j \in I$ such that $(j, s_1, \ell_1) \neq (j, s_2, \ell_2)$, where $(j, s_i, \ell_i) \in C_{\text{fin}}(f_i)$, we have

1. $s_1 > s_2$, or
2. $s_1 = s_2$ but $\ell_1 < \ell_2$

Boost captures

- capturing regular expression r
- $w \in \pi_\Sigma(\mathscr{L}(r))$
- the Boost captures of matching w with r: the largest element in

$$\{C_{\text{fin}}(f) \mid f \in \mathscr{L}(r), \pi_\Sigma(f) = w\}$$

determined by $\prec_B$

Examples

Match $w =$ "ab" with $\mathtt{a?(_1ab)_1?b?}$

Forests: $f_1 = [_0\mathtt{ab}]_0$ and $f_2 = [_0[_1\mathtt{ab}]_1]_0$

$$C(f_1, 0) = \{(0,2)\}, \quad C(f_1, 1) = \emptyset, \quad C(f_2, 0) = \{(0,2)\}, \quad C(f_2, 1) = \{(0,2)\}$$
$$C_{\text{fin}}(f_1) = \{(0,0,2), (1, \top, \bot)\}, \quad C_{\text{fin}}(f_2) = \{(0,0,2), (1,0,2)\}$$

At $j = 1$, we find $s_1 = \top$ and $s_2 = 0$, so that $s_1 > s_2$.
Therefore, $C_{\text{fin}}(f_1) \prec_B C_{\text{fin}}(f_2)$.

Match w with $\mathtt{(_1a?)_1(_2ab)_2?(_3b?)_3}$

Forests: $f_3 = [_0[_1\mathtt{a}]_1[_3\mathtt{b}]_3]_0$ and $f_4 = [_0[_1]_1[_2\mathtt{ab}]_2[_3]_3]_0$

$$C_{\text{fin}}(f_3) = \{(0,0,2), (1,0,1), (2, \top, \bot), (3,1,1)\}$$
$$C_{\text{fin}}(f_4) = \{(0,0,2), (1,0,0), (2,0,2), (3,2,0)\}$$

At $j = 1$, we find $s_3 = s_4 = 0$, $\ell_3 = 1$, and $\ell_4 = 0$, so that $\ell_4 < \ell_3$.
Therefore, $C_{\text{fin}}(f_4) \prec_B C_{\text{fin}}(f_3)$.

POSIX matching algorithm in Boost

Inside Boost:

- complete Perl-Compatible Regular Expression (PCRE) engine
- implemented by depth-first backtracking

POSIX matching algorithm:

1. • apply the PCRE-style matching engine to the input
   • record the resulting parse tree t
   • if engine rejects, then reject string
2. • apply PCRE-style matching engine to the input
   • each time it would accept on parse tree t'
      • if $C_{\text{fin}}(t) \prec_B C_{\text{fin}}(t')$, then $t \leftarrow t'$
      • reject, causing engine to backtrack
3. output t as POSIX-style result

Theorem

Boost captures can be computed in time $O(k|w||r| \log |w|)$ when matching input string w with regular expression r, and k is the number of distinct capturing indices.

Experimental results

Two testing frameworks in Python

- small one for existing matchers
- larger, extensible one for exploring different disambiguation policies

Sanity check: Almost 3 000 000 generated test cases —

- over the atoms a, b, . and the operators |, *, +, ?
- input strings over $\Sigma = \{a, b, c\}$.

Fowler's test cases

- 93 examples to test POSIX compliance
- 47 ERE; 37 without partial matching + 19 of our own
- use a Boost runner as oracle
- our formalism passed all but 2

Failed test cases

Match "x" with $\underline{(_1 . ?)_1\{2\}}$

Two possible ways of matching:

$$f_1 = [_0[_1]_1[_1 x]_1]_0$$
$$f_2 = [_0[_1 x]_1[_1]_1]_0$$

Now, $C_{\text{fin}}(f_2) \prec_B C_{\text{fin}}(f_1)$, because

$$\{(0,0,1),(1,1,0)\} \prec_B \{(0,0,1),(1,0,1)\}.$$

We prefer f_1, but Boost prefers f_2.

Similarly, matching "xxx" by $\underline{(.?.?)\{3\}}$ failed.

A bug in Boost

According to the POSIX standard:
Duplication "shall match what repeated consecutive occurrences" would match.

Therefore, $\underline{(_1 \texttt{.?})_1\{2\}} \equiv \underline{(_1 \texttt{.?})_1(_1 \texttt{.?})_1}$

Possible explanation:

- internally, Boost uses $\underline{(_1 \texttt{.?})_1(_2 \texttt{.?})_2}$
- then $C_{\text{fin}}\big([_0[_1]_1[_2\texttt{x}]_2]_0\big) \prec_B C_{\text{fin}}\big([_0[_1\texttt{x}]_1[_2]_2]_0\big)$
- but it reports $[_0[_1\texttt{x}]_1[_1]_1]_0$

Does not extend to matching "xxx" by $\underline{\texttt{(.?.?)\{3\}}}$.

We think it's a bug

- Boost has code to short-circuit duplication when it first matches an empty string
- Fine for PCRE, but not POSIX

What we've done … and what's next?

In the paper:

- capturing regular expressions + forest languages
- lots of examples
- formalisation of Boost matching semantics
- a start to the formalisation of disambiguation policies

To do:

- tackle other kinds of matching semantics
- for example, improve informal consideration of Okui–Suzuki
- disambiguation policies
  - what is possible?
  - what would be practically feasible?
  - what would be useful?

Thanks … **any questions?**