Regular expressions as types: Bit-coded regular expression parsing

Fritz Henglein

Department of Computer Science
University of Copenhagen
Email: henglein@diku.dk

WG 2.8 Meeting, Marble Falls, 2011-03-07

Joint work with Lasse Nielsen, DIKU

Regular expression

Definition (Regular expression)

A *regular expression (RE)* over finite alphabet A is an expression of the form

$$E, F ::= 0 \mid 1 \mid a \mid E|F \mid EF \mid E*$$

where $a \in A$

Used in bioinformatics, compilers (lexical analysis, control flow analysis), logic, natural language processing, program verification, protocol specification, query processing, security, XML access paths and document types, operating systems, scripting of searching, matching and substitution in texts or semi-structured data (Perl) . . .

Language interpretation of regular expressions

Definition (Language interpretation)

The *language interpretation* of a regular expression E is the set of strings $\mathcal{L}[\![E]\!]$ defined by

$$
\begin{array}{rcl}
\mathcal{L}[\![0]\!] & = & \emptyset \\
\mathcal{L}[\![1]\!] & = & \{\epsilon\} \\
\mathcal{L}[\![a]\!] & = & \{a\}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{L}[\![E|F]\!] & = & \mathcal{L}[\![E]\!] \cup \mathcal{L}[\![F]\!] \\
\mathcal{L}[\![EF]\!] & = & \mathcal{L}[\![E]\!] \odot \mathcal{L}[\![F]\!] \\
\mathcal{L}[\![E*]\!] & = & \bigcup_{i \geq 0} (\mathcal{L}[\![E]\!])^i
\end{array}
$$

where $S \odot T = \{s\,t \mid s \in S \wedge t \in T\}$, $E^0 = \{\epsilon\}, E^{i+1} = E\,E^i$.

Kleene's Theorem

Theorem (Kleene 1956)

A language is regular if and only it is denoted by a regular expression under its language interpretation.

What is regular expression "matching"?

Given regular expression and input string, return . . . what?

1. yes or no (membership testing)
2. zero or one substring matches for each regular subexpression (PCRE)
3. any finite number of substring matches for each regular subexpression (regular expression types)
4. a parse tree

What is regular expression “matching”?

1. Membership testing = language interpretation.
2. PCRE: Only one match under a Kleene star (typically the last)
3. RET: Matches under two Kleene stars not grouped
4. Parsing: Each Kleene star yields a list of matches (thus parse tree).

Note:

- Increasing structure: Lower level matching output constructible from higher level matching output, in particular from parsing.
- Classical automata theory (e.g. minimal DFA construction) only sound for membership testing.

Practice

PCRE-style programming[1]:

- Group matching: Does the RE match and where do (some of) its *sub-REs* match in the string?
- Substitution: Replace matched substrings by specified other strings
- Extensions: Backreferences, look-ahead, look-behind,...
- Lazy vs. greedy matching, possessive quantifiers, atomic grouping
- Optimization

Observe: Language interpretation (yes/no) inappropriate, need more refined interpretation

[1] in Perl and such

Example

$$\texttt{((ab)(c|d)|(abc))*}.$$

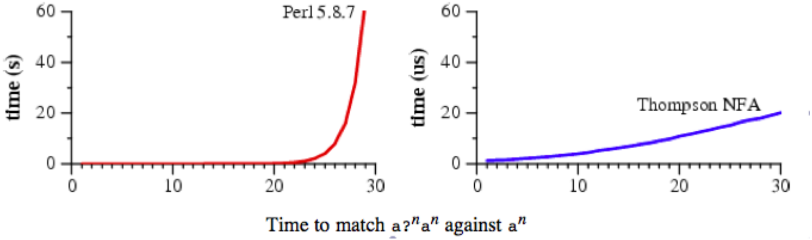
Match against $\boxed{\texttt{abdabc}}$.
For each parenthesized *group* a substring is returned.[a]

		PCRE	POSIX
\$1	=	*abc*	*abc*
\$2	=	*ab*	ϵ
\$3	=	*c*	ϵ
\$4	=	ϵ	*abc*

[a] Or special *null*-value

Intermezzo: Optimization??

Optimizing regular expressions = rewriting them to equivalent form that is more efficient for matching.[2]



Cox (2007)

- Perl-compliant regular expressions (what you get in Perl, Python, Ruby, Java) use *backtracking parsing*.
- Does not handle "problematic" regular expressions: E^* where E contains ϵ – may crash at run-time (stack overflow).

[2]Friedl, Mastering Regular Expressions, chapter 6: *Crafting an efficient expression*

Why discrepancy between theory and practice?

- Theory is *extensional*: About regular *languages*.
  - Does this string match the regular expression? Yes or no?
- Practice is *intensional*: About regular expressions as *grammars*.
  - Does this string match the regular expression and if so *how*—which parts of the string match which parts of the RE?
- Ideally: Regular expression matching = parsing + "catamorphic" processing of syntax tree
- Reality:
  - Naive backtracking matching, or
  - finite automaton + opportunistic instrumentation to get *some* parsing information (TCL (?), Laurikari 2000, Cox 2010).

Regular expression parsing

- Regular expression parsing: Construct parse tree for given string.
- Representation of parse tree: Regular expression as type

Example

Parse `abdabc` according to `((ab)(c|d)|(abc))*`.

- $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]$
- $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]$

- p_1, p_2 have *type* $\boxed{((a \times b) \times (c + d) + a \times (b \times c))\,\mathrm{list}}$.
- Compare with *regular expression* $\boxed{\texttt{((ab)(c|d)|(abc))*}}$.
- The *elements* of *type* E correspond to the *syntax trees* for strings parsed according to *regular expression* E!

Type interpretation

Definition (Type interpretation)

The *type interpretation* $\mathcal{T}[\![.]\!]$ compositionally maps a regular expression E to the corresponding simple type:

$$
\begin{array}{rcll}
\mathcal{T}[\![0]\!] & = & \emptyset & \text{empty type} \\
\mathcal{T}[\![1]\!] & = & \{()\} & \text{unit type} \\
\mathcal{T}[\![a]\!] & = & \{a\} & \text{singleton type} \\
\mathcal{T}[\![E \mid F]\!] & = & \mathcal{T}[\![E]\!] + \mathcal{T}[\![F]\!] & \text{sum type} \\
\mathcal{L}[\![E\,F]\!] & = & \mathcal{T}[\![E]\!] \times \mathcal{T}[\![F]\!] & \text{product type} \\
\mathcal{T}[\![E^*]\!] & = & \{[v_1, \ldots, v_n] \mid v_i \in \mathcal{T}[\![E]\!]\} & \text{list type}
\end{array}
$$

Flattening

Definition

The *flattening* function $\mathrm{flat}(.) : \mathrm{Val}(\mathcal{A}) \to \mathrm{Seq}(\mathcal{A})$ is defined as follows:

$$
\begin{array}{rclrcl}
\mathrm{flat}(()) & = & \epsilon & \mathrm{flat}(a) & = & a \\
\mathrm{flat}(\mathrm{inl}\, v) & = & \mathrm{flat}(v) & \mathrm{flat}(\mathrm{inr}\, w) & = & \mathrm{flat}(w)
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{flat}((v, w)) & = & \mathrm{flat}(v)\, \mathrm{flat}(w) \\
\mathrm{flat}([v_1, \ldots, v_n]) & = & \mathrm{flat}(v_1) \ldots \mathrm{flat}(v_n)
\end{array}
$$

Example

$$
\begin{array}{rcl}
\mathrm{flat}([\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]) & = & abdabc \\
\mathrm{flat}([\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]) & = & abdabc
\end{array}
$$

Regular expressions as types

Informally:

string s with syntax tree p according to *regular expression* E
$\cong$
string $\mathrm{flat}(v)$ of value v element of *simple type* E

Theorem

$$\mathcal{L}[\![E]\!] = \{\mathrm{flat}(v) \mid v \in \mathcal{T}[\![E]\!]\}$$

Membership testing versus parsing

Example

E = `((ab)(c|d)|(abc))*` E_d = `(ab(c|d))*`

- E_d is *unambiguous*: If $v, w \in \mathcal{T}[\![E_d]\!]$ and $\text{flat}(v) = \text{flat}(w)$ then $v = w$. (Each string in E_d has exactly one syntax tree.)
- E is *ambiguous*. (Recall p_1 and p_2.)
- E and E_d are *equivalent*: $\mathcal{L}[\![E]\!] = \mathcal{L}[\![E_d]\!]$
- E_d "represents" the minimal deterministic finite automaton for E.
- Matching (membership testing): Easy—use E_d.
- But: How to parse *according to E* using E_d?

Bit coding

General idea:

- Have nondeterministic machine/algorithm M with no input, generating all elements of a set
- Use sequence of choices as representation of output (modulo M)

For regular languages:

- Record binary choices for expanding a regular expression E into a particular string s.
- The sequence of choices (as bits) to drive machine to particular output s as the *bit coding* of s under E.

Bit coding: Example

Example

Recall syntax trees p_1, p_2 for *abdabc* under
$E = ((a \times b) \times (c + d) + a \times (b \times c))^*$.

- $p_1 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))]$
- $p_2 = [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]$

We can *code* them by storing *only* their $\mathrm{inl}\,, \mathrm{inr}$ occurrences:

$$
\begin{aligned}
\mathrm{code}(p_1) &= 011 \\
\mathrm{code}(p_2) &= 0100
\end{aligned}
$$

Bit decoding

There is a linear-time *polytypic* function decode that can reconstitute the syntax trees.

Theorem

$\mathrm{decode}_E(\mathrm{code}_E(v)) = v$ *for all* $v \in \mathcal{T}[\![E]\!]$.

Example

$$
\begin{array}{rcl}
\mathrm{decode}_E(011) & = & [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inr}\,(a, (b, c))] \\
\mathrm{decode}_E(0100) & = & [\mathrm{inl}\,((a, b), \mathrm{inr}\, d), \mathrm{inl}\,((a, b), \mathrm{inl}\, c)]
\end{array}
$$

Why bit coding?

Bit coding of string s under E

- represents a syntax tree of s
- takes at most as much space as $|s|$ and often a lot less (depending on E)
- can be combined with statistical compression for text compression

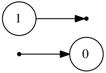
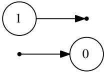
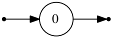
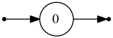
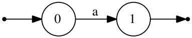
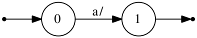
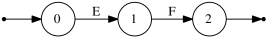
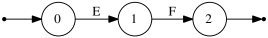
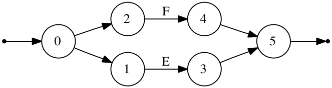
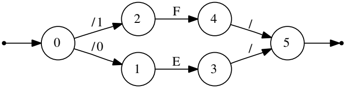
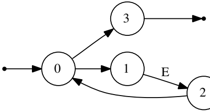
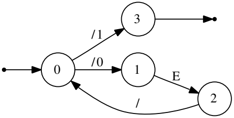
Bit coded regular expression parsing

- Problem:
  - Input: string s and regular expression E.
  - Output: (some) parse tree p such that $\mathrm{flat}(p) = s$.
- Goal: Output *bit coding* $\mathrm{code}_E(p)$ instead.
- Dual advantage:
  - Less space used for output.
  - Output faster to compute.
- How to do that? Mark the "turns" in Thompson NFA (they yield the bit coding)

DFASIM algorithm: Outline

1. RE to NFA: Build Thompson-style NFA *with suitable output bits*
2. NFA to DFA: Perform *extended* DFA construction (only for states required by input string), with (multiple) bit sequence annotations on edges
3. Traverse accepting path from *right to left* to construct bit coding by concatenating bit sequences.

Thompson-style NFA generation with output bits

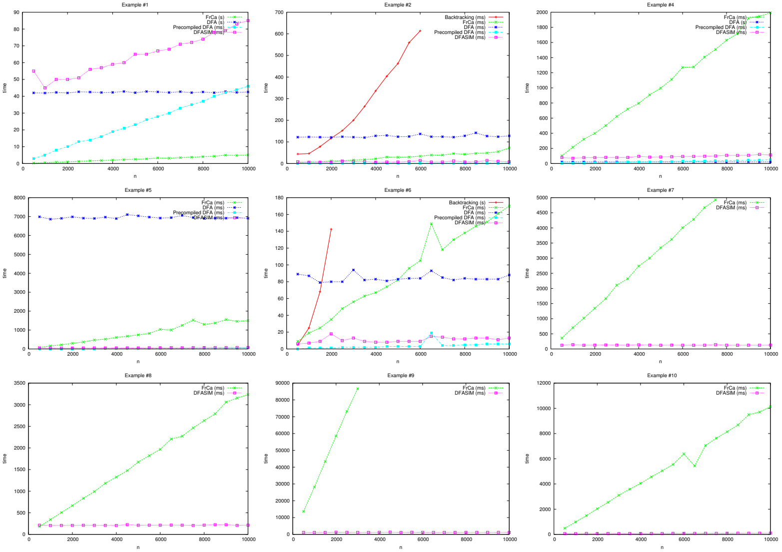
E	NFA	Extended NFA
0		
1		
a		
E F		
E | *F*		
E∗		

Benchmark examples

<table>
<tr><td>1:</td><td>\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*
([,;]\s*\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)*</td></tr>
<tr><td>2:</td><td>$?(\d{1,3},?(\d{3},?)*\d{3}(\.\d{0,2})?|\d{1,3}(\.\d{0,2})?|\.\d{1,2}?)</td></tr>
<tr><td>4:</td><td>[A-Za-z0-9](([\.\-]?[a-zA-Z0-9]+)*)@([A-Za-z0-9]+)
(([\.\-]?[a-zA-Z0-9]+)*)\.([A-Za-z][A-Za-z]+)</td></tr>
<tr><td>5:</td><td>(\w|-)+@((\w|-)+\.)+(\w|-)+</td></tr>
<tr><td>6:</td><td>[+-]?([0-9]*\.?[0-9]+|[0-9]+\.?[0-9]*)([eE][+-]?[0-9]+)?</td></tr>
<tr><td>7:</td><td>((\w|\d|\-|\.)+)@{1}(((\w|\d|\-){1,67})|((\w|\d|\-)+\.(\w|\d|\-){1,67}))
\.((([a-z]|[A-Z]|\d){2,4})(\.([a-z]|[A-Z]|\d){2})?)</td></tr>
<tr><td>8:</td><td>(([A-Za-z0-9]+ +)|([A-Za-z0-9]+\-+)|([A-Za-z0-9]+\.+)|([A-Za-z0-9]+\++))*
[A-Za-z0-9]+@((\w+\-+)|(\w+\.))*\w{1,63}\.[a-zA-Z]{2,6}</td></tr>
<tr><td>9:</td><td>(([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5}){1,25})+
([;.](([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5}){1,25})+)*</td></tr>
<tr><td>10:</td><td>((\w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*)\s*[,]{0,1}\s*)+</td></tr>
</table>

From Veanes, de Halleaux, Tillman (2010)

Benchmark experiments (without #3)



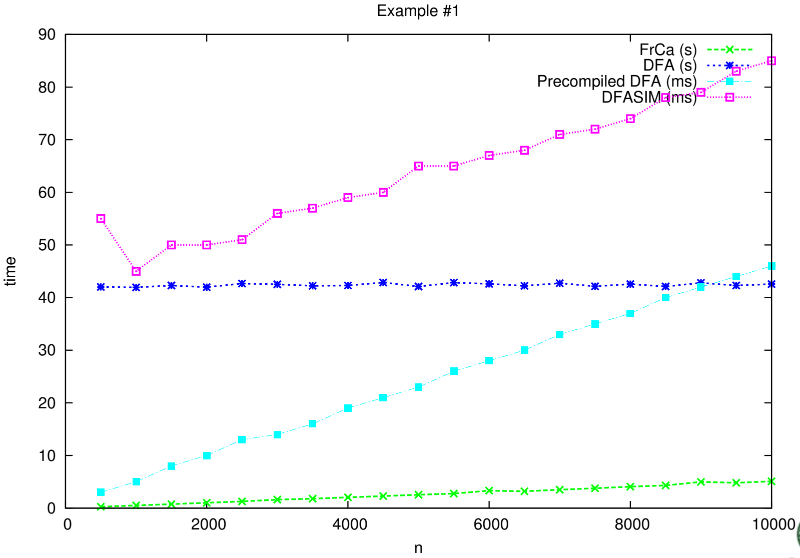
Regular expression algorithms compared

- FrCa: Based on Frisch, Cardelli (2004), right-to-left first phase, left-to-right second phase.
- DFASIM: As above.
- DFA: As DFASIM, but staged. Extended DFA for complete extended Thomson-NFA generated, before application to input.
- Precompiled DFA: As DFA, but extended DFA specialized (in C++) and compiled.
- Backtracking: PCRE-style backtracking parser.

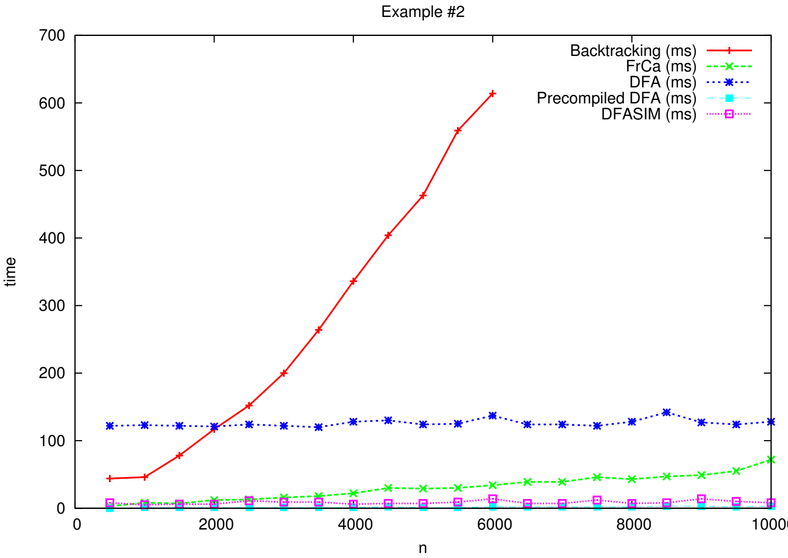
All algorithms:

- generate bit codes;
- coded in C++

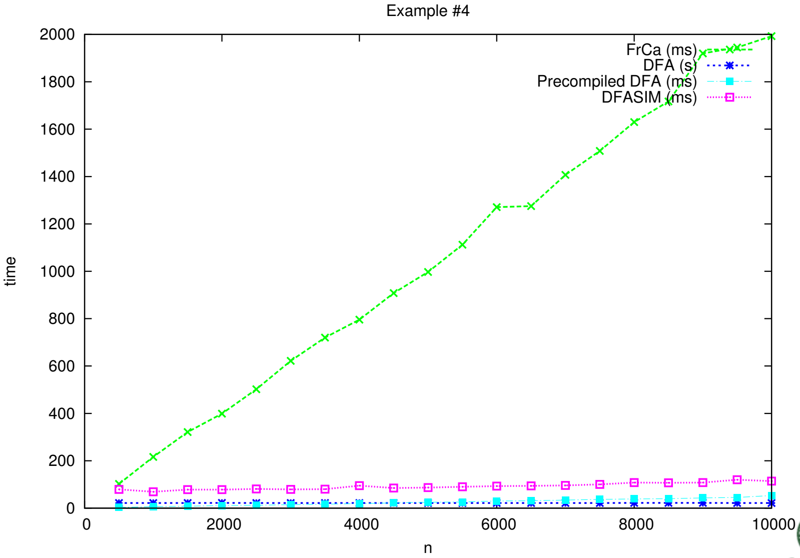
Benchmark experiment #1



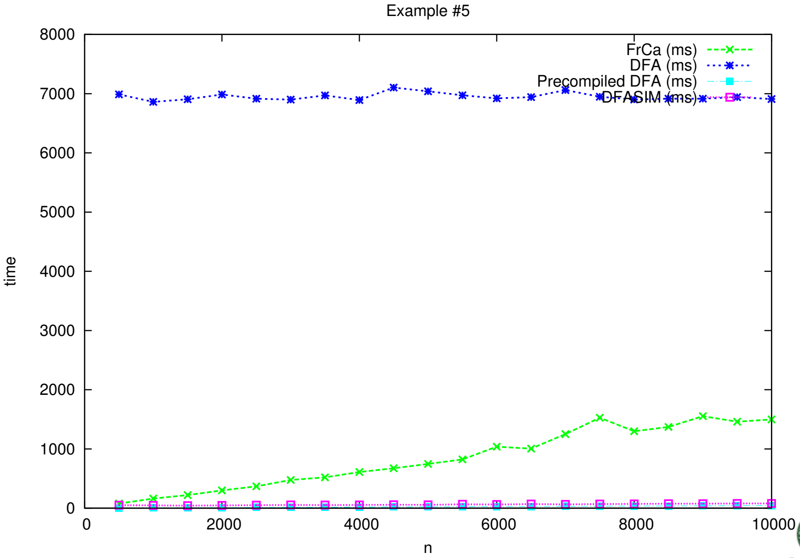
Benchmark experiment #2



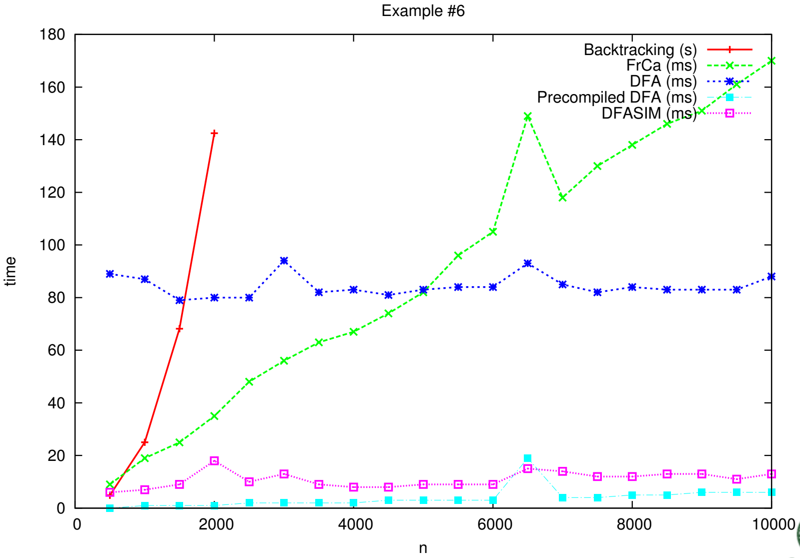
Benchmark experiment #4



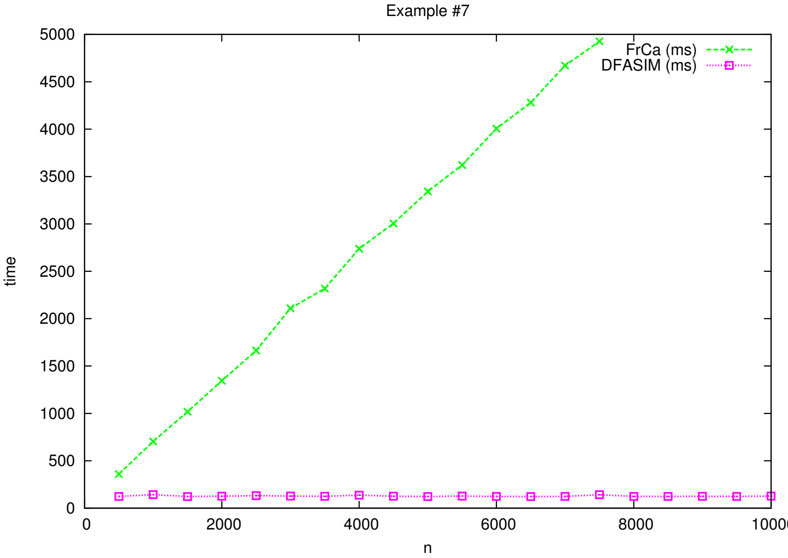
Benchmark experiment #5



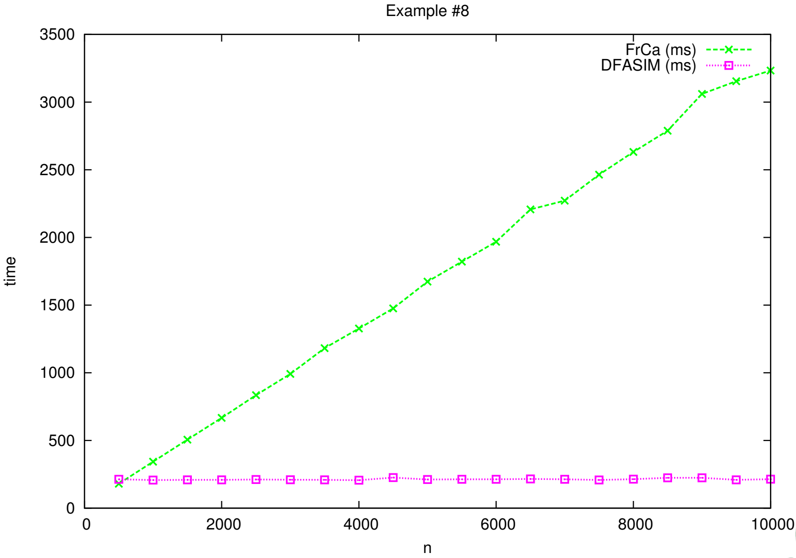
Benchmark experiment #6



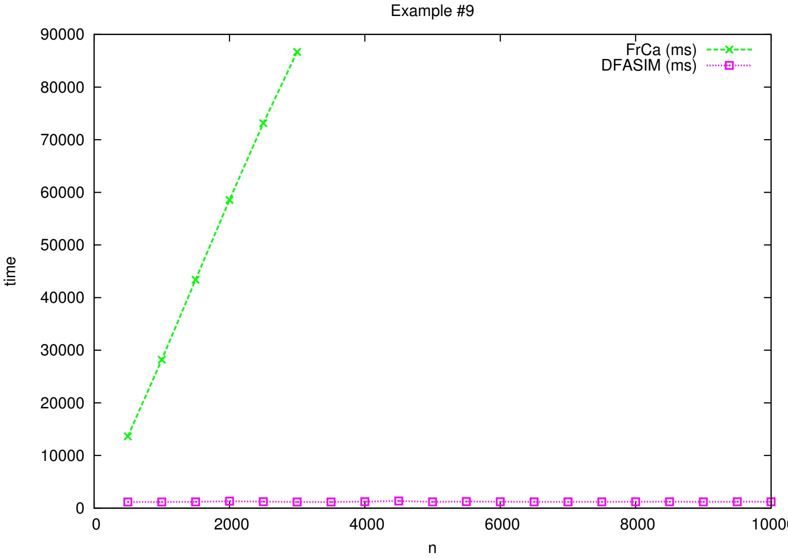
Benchmark experiment #7



Benchmark experiment #8



Benchmark experiment #9



References

- Henglein, Nielsen, "Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation", POPL 2011
- Nielsen, Henglein, "Bit-coded Regular Expression Parsing", LATA 2011

Related work

- Frisch, Cardelli (2004): Regular *types* corresponding to regular expressions, linear-time parsing for REs;
- Hosoya et al. (2000-): Regular expression types, *proper* extension of regular types (!), axiomatization of tree containment
- Aanderaa (1965), Salomaa (1966), Krob (1990), Pratt (1990), Kozen (1994, 2008), Grabmeyer (2005), Rutten et al. (2008): RE axiomatizations (extensional)
- Rutten et al. (1998-): Coalgebraic approach to systems, including finite automata, *extensional*
- Brandt/Henglein (1998): Coinduction rule and computational interpretation for recursive types
- Cameron (1988), Jansson, Jeuring (1999): Bit coding for CFGs and algebraic types
- Cox (2010): RE2 regular expression library, TCL RE library (appear to be state of the Perl/POSIX-style "regex" libraries)

Questions?

Future work

- Construction of minimal extended NFAs: All
- Regular expression parsing with projection (throwing subtrees away)
- Regular expression parsing with catamorphic postprocessing (substituting subtrees)
- Regular expression library as practical alternatives to PCRE, RE2 and Tcl, etc., with improved expressiveness, semantics and performance.