Regular-Expression Derivatives Re-Examined Analysis

Patrick Bacon-Blaber
Luís Felipe Franco Candêo Tomazini
Ethan Vynalek

Language: Java

# Introduction

The paper we read, entitled "Regular-expression derivatives re-examined", was about using regular-expression derivatives as a technique to compile regular expressions to deterministic finite-state machines (DFAs). It gives examples of how these DFAs can be made, as well as some of their improvements for larger character sets like Unicode. It also compares their implementation performance with other regex reader libraries.

# Preliminaries

In this section it is covered the notation and symbols that will be used throughout the whole article. Most of it is similar to the ones with have already seen in class with the addition of some logical operators. The following expressions can be referred as extended regular expressions, but as they are closed under boolean operations, they may also be called regular expressions (REs).

| | |
|---|---|
| $\Sigma$ | the symbols of the alphabet |
| $\Sigma^*$ | the set of all finite strings that might be formed with the symbols in $\Sigma$ |
| a, b, c, … | symbols in $\Sigma$ |
| u, v, w | the strings in $\Sigma^*$ |
| $\varepsilon$ | the empty string |
| L | the language of $\Sigma$, a set (finite or not) of strings |
| $r \cdot s$ | concatenation |
| $r^*$ | Kleene-closure |
| $r + s$ | logical or |
| $r \,\&\, s$ | logical and |
| $\neg r$ | complement |

For a regular expression r and its language $L[r] \subseteq \Sigma^*$, L[r] is a regular language if it can be described by the following rules:

$L[\varnothing] = \varnothing$
$L[\varepsilon] = \{\varepsilon\}$
$L[a] = \{a\}$
$L[r \cdot s] = \{u \cdot v \mid u \in L[r] \text{ and } v \in L[s]\}$
$L[r^*] = \{\varepsilon\} \cup L[r \cdot r^*]$
$L[r + s] = L[r] \cup L[s]$
$L[r \,\&\, s] = L[r] \cap L[s]$
$L[\neg r] = \Sigma^* \setminus L[r]$

For this paper the definition of a DFA is a quadruple, different than the one we use in class. It is defined as M = (Q, $q_0$, F, $\boldsymbol{\delta}$), where Q is a finite set of states, $q_0$($q_0 \subseteq Q$)is the initial state, F is the set of accepting states ($F \subseteq Q$) and $\boldsymbol{\delta}$ is a partial function from Q x Σ to Q.

# RE Derivatives

The main point of the article, regular expression derivatives, is talked about in the third section of the article. A derivative of a language L with respect to the symbol "a", is the language that includes only the suffixes of the strings in L that begin with "a". If L is regular, then its derivative is also regular. This is proved by showing that if L is a language that can be recognized by the DFA (Q, **Σ**, **δ**, s, F) where:

Q - set of all states

**Σ** - alphabet

$\boldsymbol{\delta}$ - a transition function $Q \times \boldsymbol{\Sigma} \rightarrow Q$

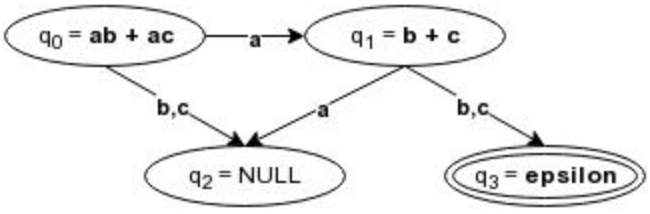
s - starting state

F - set of final states

then we can define the derivative of L can be recognized by the DFA (Q, **Σ**, $\boldsymbol{\delta}$, $\boldsymbol{\delta}$(s, a), F) which means that the DFA is regular. This can be proven inductively on the size of the derivative.

With this DFA, we can see that a string *u* is in a language *r* if the derivative with respect to *u* is *nullable*, that is, it contains ε. This means that when we are seeing if a string *u* is in a language, we compute a derivative for each character in the string, and once we hit the end we see if what we have left is nullable. When a string is not in a language, the derivative of the string reaches ø or we reach an endpoint that is not nullable.

We can create a DFA recognizer for any language of a RE. The states of this DFA are RE equivalence classes, and the transition function is the derivative function on those classes. Each state is labelled with an RE representing its equivalence class. Accepting states are those that are nullable REs, and the error state is labeled by ø. An example is given below:




# Practical DFA Construction

Section four talks about the practicality of creating these DFAs. There are three main problems with what was proposed in section three. The first is that determining whether two REs are equivalent, which is used for our equivalence classes for our states, is expensive. Next, to iterate over the symbols in **Σ** to compute the transition function is not practical for alphabets like Unicode. Lastly, a scanner generator will typically take a collection of REs as its input specification, where section three only built a DFA for one.

To deal with the first problem, we denote ≈ as an approximation of RE equivalence. Below is the definition of ≈:

| | | | |
|---|---|---|---|
| $r \,\&\, r \approx r$ | $r \,\&\, s \approx s \,\&\, r$ | $(r \,\&\, s) \,\&\, t \approx r \,\&\, (s \,\&\, t)$ | $\emptyset \,\&\, r \approx \emptyset$ |
| $\neg\emptyset \,\&\, r \approx r$ | $(r \cdot s) \cdot t \approx r \cdot (s \cdot t)$ | $\emptyset \cdot r \approx \emptyset$ | $r \cdot \emptyset \approx \emptyset$ |
| $\varepsilon \cdot r \approx r$ | $r \cdot \varepsilon \approx r$ | $r + r \approx r$ | $r + s \approx s + r$ |
| $(r + s) + t \approx r + (s + t)$ | $\neg\emptyset + r \approx \neg\emptyset$ | $\emptyset + r \approx r$ | $(r^*)^* \approx r^*$ |
| $\varepsilon^* \approx \varepsilon$ | $\emptyset^* \approx \varepsilon$ | $\neg(\neg r) \approx r$ | |

If $r \approx s$, then they are equivalent. By using these similar equations, we can guarantee termination and and most of the time get the minimal DFA, which makes our determination of two REs being equivalent much quicker and more realistic..

The second problem that was brought up was the size of the alphabets that have to be iterated over. To narrow down the alphabet, we divide it's contents into derivative classes. We say that two symbols are in the same derivative class if the derivative with respect to those two symbols on the regular expression is the same. To quickly find these derivative classes, we define a function C:

$C(\varepsilon) = \{\mathbf{\Sigma}\}$

$C(L) = \{L, \mathbf{\Sigma} \setminus L\}$

$C(r \cdot s) = C(r)$ if r is not nullable, $C(r) \wedge C(s)$ otherwise

$C(r + s) = C(r \,\&\, s) = C(r) \wedge C(s)$

$C(r^*) = C(r)$

$C(\neg r) = C(r)$

Using this function on a regular expression, we can get the derivative classes of that language quickly. This allows us to iterate over larger alphabets efficiently.

The last problem was that the section three DFA couldn't handle multiple REs in parallel. To deal with this, we label the DFA states with *regular vectors* instead of regular expressions. A regular vector is an n-tuple of REs, $R = (r_1, r_2, \ldots, r_n)$. The derivative of a regular vector is defined as a regular vector that contains the derivative of each regular expression it contained. With these changes, multiple REs can be can be handled at once.

# Experience

Section 5 compares the work discussed in the paper to the work done by some others. The paper claims that their use of a derivatives to compute DFAs makes them much smaller. Out of all the lexer's that were compared to, the derivative based lexer had less states. In most cases, the derivative based one had the minimal number of states. They also talk about how their character classes made the number of derivatives computed minimal. There were only two cases where their algorithm wasn't perfect, and it only computed 5.4% and 6.2% more derivatives than necessary. If the ASCII alphabet was the input, then their algorithm would compute only 2%-4% of the possible derivatives, leading them to the conclusion that their character classes provide a significant benefit in the construction of DFAs, even if the alphabet is small.

## Related Work

In discussing scientific endeavors similar in nature to their own, the authors appear to take care to distinguish their own work from the works of others, highlighting key differences to promote a sense of uniqueness. We did notice, however, that most of the comparisons they draw between their own work and the works of others only involve a dichotomous explanation of the strengths of their technique and the relative weaknesses or disadvantages of the strategies employed by others. Whether this is a true conveyance of the factual differences between works or simply embellishment on the part of the authors remains to be seen.

The authors first mention (fleetingly) the occasional use of RE derivatives in XML validation tasks that don't build automata. They go on to describe two others systems that employ derivatives in DFA construction: the early stages of the *Esterel* language (derivatives abandoned for memory reasons), and a program trace monitoring system that used a technique called "circular coinduction" to generate minimal DFAs by testing full RE equivalence. The authors describe the latter as seemingly less practical than their own, but in this instance their argument is supported by a discussion of the system's shortcomings.

The paper goes on to mention Berry and Sethi's (1986) demonstration of how McNaughton and Yamada's (1960) algorithm can be derived by a DFA construction algorithm employing derivatives, and notes the differences between their work and Brzozowski's (1964) derivative algorithm. Furthermore, they bring up Berry and Sethi's observations on the difficulties involved in adapting algorithms based on marked symbols to support *complement* and *intersection* operations, immediately followed by an example of a system (DMS; Baxter et. al., 2004) that does just this by converting between DFAs and NFAs.

## Scientific Contributions

The main scientific contribution of this article is to show a superior way of generating scanners from REs. They state that RE derivatives provide a direct RE to DFA translation, they support extended REs with almost no extra work, and the scanners that are generated are optimal in the number of states most of the time.

The extension of REs to include boolean operations enabling tools such as complementation can be enormously beneficial to its expressive capabilities. Given as an example in the paper is the RE matching "C-style comments:"

$$/*\sim(\Sigma^* * /\Sigma^*)*/$$

This RE employs the complement operator to effectively recognize strings that match the format specified as comments in C and other languages. The paper states that "expressing this pattern without the complement operator is more cumbersome…" and gives the (lengthy) RE required to match C-style comment-like strings without "~". It goes on to mention the complexity that would be involved in the implementation of RE subtraction, were it not for the extended RE framework.

Also of note was the effectiveness of the system at generating state machines of minimal size. The authors, in discussing their testing procedures, give 14 pre-existing `ml-lex` specifications for various languages, and note "In most cases, the RE derivative method produced a smaller state

machine...Furthermore, `ml-ulex` produces the minimal state machine for every example except Russo [a specification for mining system logs for interesting events]...” In the following paragraphs, they give an in-depth explanation of the derivative algorithm and why it produces smaller DFAs.

## Significant Results

This paper has as its main purpose to show the technique of regular-expression derivatives, which even though it is an old approach in the field of functional programming, not to many computer scientists know it. It was pointed out the advantages and the occasions when using a RE derivative is a more efficient way of solving the problem.

Another aspect covered by this paper is to demonstrate that RE derivatives are a much better procedure for generating scanners from regular-expressions. They also discussed about implementations of a scanner generator based on regular expressions, which includes support for large character sets.

## Implementation

For the next phase of the project we are going to implement two parts covered in the paper: *Using derivatives for RE matching* and *Using derivatives for DFA construction*. In order to do that firstly we need to implement the derivative function for the regular expressions.