POSIX Regular Expression Matching and Lexing

Annonymous

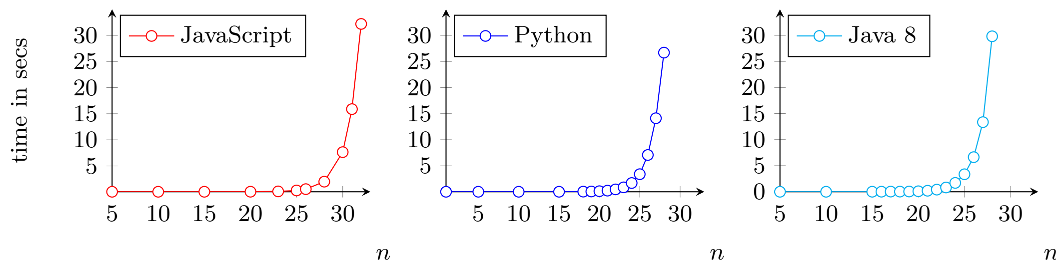
Abstract

Brzozowski introduced in 1964 a beautifully simple algorithm for regular expression matching based on the notion of derivatives of regular expressions. In 2014, Sulzmann and Lu extended this algorithm to not just give a YES/NO answer for whether or not a regular expression matches a string, but in case it matches also *how* it matches the string. This is important for applications such as lexing (tokenising a string). The problem is to make the algorithm by Sulzmann and Lu fast on all inputs without breaking its correctness.

1 Introduction

This PhD-project is about regular expression matching and lexing. Given the maturity of this topic, the reader might wonder: Surely, regular expressions must have already been studied to death? What could possibly be *not* known in this area? And surely all implemented algorithms for regular expression matching are blindingly fast?

Unfortunately these preconceptions are not supported by evidence: Take for example the regular expression $(a^*)^*\,b$ and ask whether strings of the form $aa..a$ match this regular expression. Obviously they do not match—the expected b in the last position is missing. One would expect that modern regular expression matching engines can find this out very quickly. Alas, if one tries this example in JavaScript, Python or Java 8 with strings like 28 a's, one discovers that this decision takes around 30 seconds and takes considerably longer when adding a few more a's, as the graphs below show:



Graphs: Runtime for matching $(a^*)^*\,b$ with strings of the form $\underbrace{aa..a}_{n}$.

These are clearly abysmal and possibly surprising results. One would expect these systems doing much better than that—after all, given a DFA and a string, whether a string is matched by this DFA should be linear.

Admittedly, the regular expression $(a^*)^*\,b$ is carefully chosen to exhibit this "exponential behaviour". Unfortunately, such regular expressions are not just a few "outliers", but actually they are frequent enough that a separate name has been created for them—*evil regular expressions*. In empiric work, Davis et al report that they have found thousands of such evil regular expressions in the JavaScript and Python ecosystems [**?**].

This exponential blowup sometimes causes real pain in "real life": for example one evil regular expression brought on 20 July 2016 the webpage Stack Exchange to its knees. In this instance, a regular expression intended to just trim white spaces from the beginning and the end of a line actually consumed massive amounts of CPU-resources and because of this the web servers ground to a halt. This happened when a post with 20,000 white spaces

licensed under Creative Commons License CC-BY
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

was submitted, but importantly the white spaces were neither at the beginning nor at the end. As a result, the regular expression matching engine needed to backtrack over many choices.

The underlying problem is that many "real life" regular expression matching engines do not use DFAs for matching. This is because they support regular expressions that are not covered by the classical automata theory, and in this more general setting there are quite a few research questions still unanswered and fast algorithms still need to be developed.

There is also another under-researched problem to do with regular expressions and lexing, i.e. the process of breaking up strings into sequences of tokens according to some regular expressions. In this setting one is not just interested in whether or not a regular expression matches a string, but if it matches also in *how* it matches the string. Consider for example a regular expression r_{key} for recognising keywords such as *if*, *then* and so on; and a regular expression r_{id} for recognising identifiers (say, a single character followed by characters or numbers). One can then form the compound regular expression $(r_{key} + r_{id})^*$ and use it to tokenise strings. But then how should the string *iffoo* be tokenised? It could be tokenised as a keyword followed by an identifier, or the entire string as a single identifier. Similarly, how should the string *if* be tokenised? Both regular expressions, r_{key} and r_{id}, would "fire"—so is it an identifier or a keyword? While in applications there is a well-known strategy to decide these questions, called POSIX matching, only relatively recently precise definitions of what POSIX matching actually means have been formalised [**?**, **?**, **?**]. Roughly, POSIX matching means to match the longest initial substring and possible ties are solved according to some priorities attached to the regular expressions (e.g. keywords have a higher priority than identifiers). This sounds rather simple, but according to Grathwohl et al [**?**, Page 36] this is not the case. They wrote:

> *"The POSIX strategy is more complicated than the greedy because of the dependence on information about the length of matched strings in the various subexpressions."*

This is also supported by evidence collected by Kuklewicz [**?**] who noticed that a number of POSIX regular expression matchers calculate incorrect results.

Our focus is on an algorithm introduced by Sulzmann and Lu in 2014 for regular expression matching according to the POSIX strategy [**?**]. Their algorithm is based on an older algorithm by Brzozowski from 1964 where he introduced the notion of derivatives of regular expressions [**?**]. We shall briefly explain the algorithms next.

2 The Algorithms by Brzozowski, and Sulzmann and Lu

Suppose regular expressions are given by the following grammar (for the moment ignore the grammar for values on the right-hand side):

Regular Expressions			**Values**		
r	$::=$	$\mathbf{0}$	v	$::=$	
	$\mid$	$\mathbf{1}$			Empty
	$\mid$	c		$\mid$	$\mathit{Char}(c)$
	$\mid$	$r_1 \cdot r_2$		$\mid$	$\mathit{Seq}\, v_1\, v_2$
	$\mid$	$r_1 + r_2$		$\mid$	$\mathit{Left}(v)$
				$\mid$	$\mathit{Right}(v)$
	$\mid$	r^*		$\mid$	$\mathit{Stars}\,[v_1, \ldots v_n]$

The intended meaning of the regular expressions is as usual: $\mathbf{0}$ cannot match any string, $\mathbf{1}$ can match the empty string, the character regular expression c can match the character c,

and so on. The brilliant contribution by Brzozowski is the notion of *derivatives* of regular expressions. The idea behind this notion is as follows: suppose a regular expression r can match a string of the form $c :: s$ (that is a list of characters starting with c), what does the regular expression look like that can match just s? Brzozowski gave a neat answer to this question. He defined the following operation on regular expressions, written $r\backslash c$ (the derivative of r w.r.t. the character c):

$$
\begin{array}{lcl}
\mathbf{0}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0}\\
\mathbf{1}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0}\\
d\backslash c & \stackrel{\text{def}}{=} & \textit{if}\ c = d\ \textit{then}\ \mathbf{1}\ \textit{else}\ \mathbf{0}\\
(r_1 + r_2)\backslash c & \stackrel{\text{def}}{=} & r_1\backslash c\ +\ r_2\backslash c\\
(r_1 \cdot r_2)\backslash c & \stackrel{\text{def}}{=} & \textit{if}\, \textit{nullable}(r_1)\\
 & & \textit{then}\ (r_1\backslash c)\cdot r_2\ +\ r_2\backslash c\\
 & & \textit{else}\ (r_1\backslash c)\cdot r_2\\
(r^*)\backslash c & \stackrel{\text{def}}{=} & (r\backslash c)\cdot r^*
\end{array}
$$

In this definition *nullable*(_) stands for a simple recursive function that tests whether a regular expression can match the empty string (its definition is omitted). Assuming the classic notion of a *language* of a regular expression, written $L(_)$, the main property of the derivative operation is that

$$c :: s \in L(r) \text{ holds if and only if } s \in L(r\backslash c).$$

The beauty of derivatives is that they lead to a really simple regular expression matching algorithm: To find out whether a string s matches with a regular expression r, build the derivatives of r w.r.t. (in succession) all the characters of the string s. Finally, test whether the resulting regular expression can match the empty string. If yes, then r matches s, and no in the negative case. For us the main advantage is that derivatives can be straightforwardly implemented in any functional programming language, and are easily definable and reasoned about in theorem provers—the definitions just consist of inductive datatypes and simple recursive functions. Moreover, the notion of derivatives can be easily generalised to cover extended regular expression constructors such as the not-regular expression, written $\neg\, r$, or bounded repetitions (for example $r^{\{n\}}$ and $r^{\{n..m\}}$), which cannot be so straightforwardly realised within the classic automata approach.

One limitation, however, of Brzozowski's algorithm is that it only produces a YES/NO answer for whether a string is being matched by a regular expression. Sulzmann and Lu [**?**] extended this algorithm to allow generation of an actual matching, called a *value*—see the grammar above for its definition. Assuming a regular expression matches a string, values encode the information of *how* the string is matched by the regular expression—that is, which part of the string is matched by which part of the regular expression. To illustrate this consider the string xy and the regular expression $(x + (y + xy))^*$. We can view this regular expression as a tree and if the string xy is matched by two Star "iterations", then the x is matched by the left-most alternative in this tree and the y by the right-left alternative. This suggests to record this matching as

$$\textit{Stars}\,[\textit{Left}\,(\textit{Char}\,x), \textit{Right}(\textit{Left}(\textit{Char}\,y))]$$

where *Stars* records how many iterations were used; and *Left*, respectively *Right*, which alternative is used. The value for matching xy in a single "iteration", i.e. the POSIX value, would look as follows

$$Stars\,[Seq\,(Char\,x)\,(Char\,y)]$$

where *Stars* has only a single-element list for the single iteration and *Seq* indicates that xy is matched by a sequence regular expression.

The contribution of Sulzmann and Lu is an extension of Brzozowski's algorithm by a second phase (the first phase being building successive derivatives). In this second phase, for every successful match the corresponding POSIX value is computed. As mentioned before, from this value we can extract the information *how* a regular expression matched a string. We omit the details here on how Sulzmann and Lu achieved this (see [**?**]). Rather, we shall focus next on the process of simplification of regular expressions, which is needed in order to obtain *fast* versions of the Brzozowski's, and Sulzmann and Lu's algorithms. This is where the PhD-project hopes to advance the state-of-the-art.

3 Simplification of Regular Expressions

The main drawback of building successive derivatives according to Brzozowski's definition is that they can grow very quickly in size. This is mainly due to the fact that the derivative operation generates often "useless" $\mathbf{0}$s and $\mathbf{1}$s in derivatives. As a result, if implemented naively both algorithms by Brzozowski and by Sulzmann and Lu are excruciatingly slow. For example when starting with the regular expression $(a + aa)^*$ and building 12 successive derivatives w.r.t. the character a, one obtains a derivative regular expression with more than 8000 nodes (when viewed as a tree). Operations like derivative and *nullable* need to traverse such trees and consequently the bigger the size of the derivative the slower the algorithm. Fortunately, one can simplify regular expressions after each derivative step. Various simplifications of regular expressions are possible, such as the simplifications of $\mathbf{0}+r$, $r+\mathbf{0}$, $\mathbf{1}\cdot r$, $r\cdot\mathbf{1}$, and $r+r$ to just r. These simplifications do not affect the answer for whether a regular expression matches a string or not, but fortunately also do not affect the POSIX strategy of how regular expressions match strings—although the latter is much harder to establish. Some initial results in this regard have been obtained in [**?**]. However, what has not been achieved yet is a very tight bound for the size. Such a tight bound is suggested by work of Antimirov who proved that (partial) derivatives can be bound by the number of characters contained in the initial regular expression [**?**]. We believe, and have generated test data, that a similar bound can be obtained for the derivatives in Sulzmann and Lu's algorithm. Let us give some details about this next.

We first followed Sulzmann and Lu's idea of introducing *annotated regular expressions* [**?**]. They are defined by the following grammar:

$$\begin{array}{lcl} a & ::= & ZERO\\ & \mid & ONE\;\; bs\\ & \mid & CHAR\;\; bs\,c\\ & \mid & ALTS\;\; bs\,as\\ & \mid & SEQ\;\; bs\,a_1\,a_2\\ & \mid & STAR\;\; bs\,a \end{array}$$

where bs stands for bitsequences, and as (in $ALTS$) for a list of annotated regular expressions. These bitsequences encode information about the (POSIX) value that should be generated by the Sulzmann and Lu algorithm. There are operations that can transform the usual (un-annotated) regular expressions into annotated regular expressions, and there are operations for encoding/decoding values to or from bitsequences. For example the encoding function for values is defined as follows:

$$
\begin{array}{lcl}
\textit{code}(\textit{Empty}) & \stackrel{\text{def}}{=} & [] \\
\textit{code}(\textit{Char}\, c) & \stackrel{\text{def}}{=} & [] \\
\textit{code}(\textit{Left}\, v) & \stackrel{\text{def}}{=} & Z :: \textit{code}(v) \\
\textit{code}(\textit{Right}\, v) & \stackrel{\text{def}}{=} & S :: \textit{code}(v) \\
\textit{code}(\textit{Seq}\, v_1\, v_2) & \stackrel{\text{def}}{=} & \textit{code}(v_1) \,@\, \textit{code}(v_2) \\
\textit{code}(\textit{Stars}\, []) & \stackrel{\text{def}}{=} & [S] \\
\textit{code}(\textit{Stars}\, (v :: vs)) & \stackrel{\text{def}}{=} & Z :: \textit{code}(v) \;@\; \textit{code}(\textit{Stars}\, vs)
\end{array}
$$

where Z and S are arbitrary names for the "bits" in the bitsequences. Although this encoding is "lossy" in the sense of not recording all details of a value, Sulzmann and Lu have defined the decoding function such that with additional information (namely the corresponding regular expression) a value can be precisely extracted from a bitsequence.

The main point of the bitsequences and annotated regular expressions is that we can apply rather aggressive (in terms of size) simplification rules in order to keep derivatives small. We have developed such "aggressive" simplification rules and generated test data that show that the expected bound can be achieved. Obviously we could only cover partially the search space as there are infinitely many regular expressions and strings. One modification we introduced is to allow a list of annotated regular expressions in the *ALTS* constructor. This allows us to not just delete unnecessary **0**s and **1**s from regular expressions, but also unnecessary "copies" of regular expressions (very similar to simplifying $r + r$ to just r, but in a more general setting). Another modification is that we use simplification rules inspired by Antimirov's work on partial derivatives. They maintain the idea that only the first "copy" of a regular expression in an alternative contributes to the calculation of a POSIX value. All subsequent copies can be prunned from the regular expression.

We are currently engaged with proving that our simplification rules actually do not affect the POSIX value that should be generated by the algorithm according to the specification of a POSIX value and furthermore that our derivatives stay small for all derivatives. For this proof we use the theorem prover Isabelle. Once completed, this result will advance the state-of-the-art: Sulzmann and Lu wrote in their paper [**?**] about the bitcoded "incremental parsing method" (that is the matching algorithm outlined in this section):

> *"Correctness Claim: We further claim that the incremental parsing method in Figure 5 in combination with the simplification steps in Figure 6 yields POSIX parse trees. We have tested this claim extensively by using the method in Figure 3 as a reference but yet have to work out all proof details."*

We would settle the correctness claim and furthermore obtain a much tighter bound on the sizes of derivatives. The result is that our algorithm should be correct and faster on all inputs. The original blow-up, as observed in JavaScript, Python and Java, would be excluded from happening in our algorithm.

4 Conclusion

In this PhD-project we are interested in fast algorithms for regular expression matching. While this seems to be a "settled" area, in fact interesting research questions are popping up as soon as one steps outside the classic automata theory (for example in terms of what kind of regular expressions are supported). The reason why it is interesting for us to look at the derivative approach introduced by Brzozowski for regular expression matching, and

then much further developed by Sulzmann and Lu, is that derivatives can elegantly deal with some of the regular expressions that are of interest in “real life”. This includes the not-regular expression, written $\neg\, r$ (that is all strings that are not recognised by r), but also bounded regular expressions such as $r^{\{n\}}$ and $r^{\{n..m\}}$). There is also hope that the derivatives can provide another angle for how to deal more efficiently with back-references, which are one of the reasons why regular expression engines in JavaScript, Python and Java choose to not implement the classic automata approach of transforming regular expressions into NFAs and then DFAs—because we simply do not know how such back-references can be represented by DFAs.