POSIX Regular Expression Matching and Lexing

Chengsong Tan

King's College London
London, UK
chengsong.tan@kcl.ac.uk

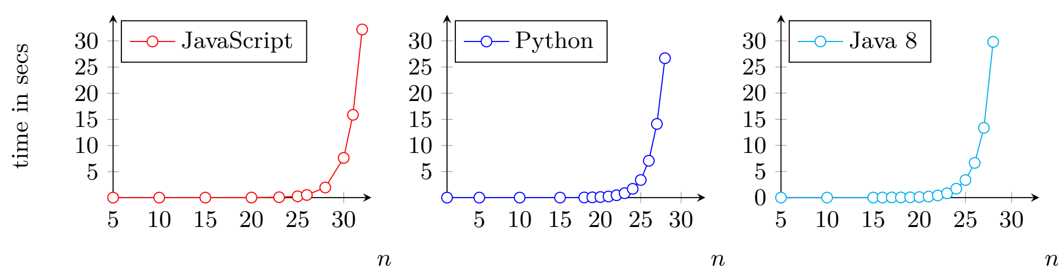
Abstract

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 does also answers with *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. We have already developed some simplification rules for this, but have not yet proved that they preserve the correctness of the algorithm. We also have not yet looked at extended regular expressions, such as bounded repetitions, negation and back-references.

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?

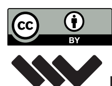
Unfortunately these preconceptions are not supported by evidence: Take for example the regular expression $(a^*)^*\,b$ and ask whether strings of the form $aa..a$ match this regular expression. Obviously this is not the case—the expected b in the last position is missing. One would expect that modern 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 to do much better than that—after all, given a DFA and a string, deciding whether a string is matched by this DFA should be linear in terms of the size of the regular expression and the string?

Admittedly, the regular expression $(a^*)^*\,b$ is carefully chosen to exhibit this super-linear behaviour. But unfortunately, such regular expressions are not just a few outliers. They are


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

actually frequent enough to have a separate name 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 [**?**]. Static analysis approach that is both sound and complete exists[**?**], but the running time on certain examples in the RegExLib and Snort regular expressions libraries is unacceptable. Therefore the problem of efficiency still remains.

This superlinear blowup in matching algorithms sometimes causes considerable grief in real life: for example on 20 July 2016 one evil regular expression brought the webpage Stack Exchange to its knees.[1] 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—causing web servers to grind to a halt. This happened when a post with 20,000 white spaces 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. In this example, the time needed to process the string was $O(n^2)$ with respect to the string length. This quadratic overhead was enough for the homepage of Stack Exchange to respond so slowly that the load balancer assumed there must be some attack and therefore stopped the servers from responding to any requests. This made the whole site become unavailable. Another very recent example is a global outage of all Cloudflare servers on 2 July 2019. A poorly written regular expression exhibited exponential behaviour and exhausted CPUs that serve HTTP traffic. Although the outage had several causes, at the heart was a regular expression that was used to monitor network traffic.[2]

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 (for example how to treat efficiently bounded repetitions, negation and back-references).

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 also in *how*. 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 [**?**, **?**, **?**]. Such a definition has also been given by Sulzmann and Lu [**?**], but the corresponding correctness proof turned out to be faulty [**?**]. Roughly, POSIX matching means matching the longest initial substring. In the case of a tie, the initial sub-match is chosen 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

---

[1] https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016

[2] https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/

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 in this project 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 this algorithm next.

2 The Algorithm by Brzozowski based on Derivatives of Regular Expressions

Suppose (basic) regular expressions are given by the following grammar:

$$r ::= \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r^*$$

The intended meaning of the constructors is as follows: $\mathbf{0}$ 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 ingenious 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 started with the definition of *nullable*:

$$\begin{array}{lcl}
\textit{nullable}(\mathbf{0}) & \stackrel{\text{def}}{=} & \textit{false} \\
\textit{nullable}(\mathbf{1}) & \stackrel{\text{def}}{=} & \textit{true} \\
\textit{nullable}(c) & \stackrel{\text{def}}{=} & \textit{false} \\
\textit{nullable}(r_1 + r_2) & \stackrel{\text{def}}{=} & \textit{nullable}(r_1) \vee \textit{nullable}(r_2) \\
\textit{nullable}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \textit{nullable}(r_1) \wedge \textit{nullable}(r_2) \\
\textit{nullable}(r^*) & \stackrel{\text{def}}{=} & \textit{true}
\end{array}$$

This function simply tests whether the empty string is in $L(r)$. He then defined the following operation on regular expressions, written $r\backslash c$ (the derivative of r w.r.t. the character c):

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

The main property of the derivative operation is that

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

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. For the moment however, we focus only on the usual basic regular expressions.

Now if we want to find out whether a string s matches with a regular expression r, we can 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. To implement this idea we can generalise the derivative operation to strings like this:

$$\begin{array}{lcl} r\backslash(c :: s) & \stackrel{\text{def}}{=} & (r\backslash c)\backslash s \\ r\backslash[\,] & \stackrel{\text{def}}{=} & r \end{array}$$

and then define as regular-expression matching algorithm:

$$\textit{match}\; s\; r \stackrel{\text{def}}{=} \textit{nullable}(r\backslash s)$$

This algorithm looks graphically as follows:

$$r_0 \xrightarrow{\backslash c_0} r_1 \xrightarrow{\backslash c_1} r_2 \dashrightarrow r_n \xrightarrow{\textit{nullable}?} \text{YES/NO} \qquad (1)$$

where we start with a regular expression r_0, build successive derivatives until we exhaust the string and then use *nullable* to test whether the result can match the empty string. It can be relatively easily shown that this matcher is correct (that is given an $s = c_0 ... c_{n-1}$ and an r_0, it generates YES if and only if $s \in L(r_0)$).

3 Values and the Algorithm by Sulzmann and Lu

One limitation 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* or sometimes also *lexical value*. These values and regular expressions correspond to each other as illustrated in the following table:

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

No value corresponds to $\mathbf{0}$; *Empty* corresponds to $\mathbf{1}$; *Char* to the character regular expression; *Seq* to the sequence regular expression and so on. The idea of values is to encode a kind of lexical value for how the sub-parts of a regular expression match the sub-parts of a string. To see this, suppose a *flatten* operation, written $|v|$ for values. We can use this function to

extract the underlying string of a value v. For example, $|Seq\,(Char\;x)\,(Char\;y)|$ is the string xy. Using flatten, we can describe how values encode lexical values: $Seq\,v_1\,v_2$ encodes a tree with two children nodes that tells how the string $|v_1|@|v_2|$ matches the regex $r_1 \cdot r_2$ whereby r_1 matches the substring $|v_1|$ and, respectively, r_2 matches the substring $|v_2|$. Exactly how these two are matched is contained in the children nodes v_1 and v_2 of parent *Seq*.

To give a concrete example of how values work, 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

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

where *Stars* $[\ldots]$ records all the iterations; 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—see (**??**)). In this second phase, a POSIX value is generated in case the regular expression matches the string. Pictorially, the Sulzmann and Lu algorithm is as follows:

$$\begin{array}{ccccccc}
r_0 & \xrightarrow{\backslash c_0} & r_1 & \xrightarrow{\backslash c_1} & r_2 & \dashrightarrow & r_n \\
\downarrow & & \downarrow & & \downarrow & & \downarrow{\scriptstyle mkeps} \\
v_0 & \xleftarrow[inj_{r_0}\,c_0]{} & v_1 & \xleftarrow[inj_{r_1}\,c_1]{} & v_2 & \dashleftarrow & v_n
\end{array} \tag{2}$$

For convenience, we shall employ the following notations: the regular expression we start with is r_0, and the given string s is composed of characters $c_0 c_1 \ldots c_{n-1}$. In the first phase from the left to right, we build the derivatives r_1, r_2, ... according to the characters c_0, c_1 until we exhaust the string and obtain the derivative r_n. We test whether this derivative is *nullable* or not. If not, we know the string does not match r and no value needs to be generated. If yes, we start building the values incrementally by *injecting* back the characters into the earlier values $v_n, \ldots, v_0$. This is the second phase of the algorithm from the right to left. For the first value v_n, we call the function *mkeps*, which builds the lexical value for how the empty string has been matched by the (nullable) regular expression r_n. This function is defined as

$$\begin{array}{lcl}
mkeps(\mathbf{1}) & \stackrel{\text{def}}{=} & Empty \\
mkeps(r_1 + r_2) & \stackrel{\text{def}}{=} & if\;\; nullable(r_1) \\
 & & then\;\; Left(mkeps(r_1)) \\
 & & else\;\; Right(mkeps(r_2)) \\
mkeps(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & Seq\,(mkeps\,r_1)\,(mkeps\,r_2) \\
mkeps(r^*) & \stackrel{\text{def}}{=} & Stars\,[]
\end{array}$$

There are no cases for $\mathbf{0}$ and c, since these regular expression cannot match the empty string. Note also that in case of alternatives we give preference to the regular expression on the left-hand side. This will become important later on about what value is calculated.

After the *mkeps*-call, we inject back the characters one by one in order to build the lexical value v_i for how the regex r_i matches the string s_i ($s_i = c_i \ldots c_{n-1}$) from the previous lexical value v_{i+1}. After injecting back n characters, we get the lexical value for how r_0 matches s. For this Sulzmann and Lu defined a function that reverses the "chopping off" of characters during the derivative phase. The corresponding function is called *injection*, written *inj*; it takes three arguments: the first one is a regular expression r_{i-1}, before the character is chopped off, the second is a character c_{i-1}, the character we want to inject and the third argument is the value v_i, into which one wants to inject the character (it corresponds to the regular expression after the character has been chopped off). The result of this function is a new value. The definition of *inj* is as follows:

$$\begin{array}{ll}
inj\,(c)\,c\,Empty & \stackrel{\text{def}}{=} Char\,c\\
inj\,(r_1 + r_2)\,c\,Left(v) & \stackrel{\text{def}}{=} Left(inj\,r_1\,c\,v)\\
inj\,(r_1 + r_2)\,c\,Right(v) & \stackrel{\text{def}}{=} Right(inj\,r_2\,c\,v)\\
inj\,(r_1 \cdot r_2)\,c\,Seq(v_1, v_2) & \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2)\\
inj\,(r_1 \cdot r_2)\,c\,Left(Seq(v_1, v_2)) & \stackrel{\text{def}}{=} Seq(inj\,r_1\,c\,v_1, v_2)\\
inj\,(r_1 \cdot r_2)\,c\,Right(v) & \stackrel{\text{def}}{=} Seq(mkeps(r_1), inj\,r_2\,c\,v)\\
inj\,(r^*)\,c\,Seq(v, Stars\,vs) & \stackrel{\text{def}}{=} Stars((inj\,r\,c\,v) :: vs)
\end{array}$$

This definition is by recursion on the "shape" of regular expressions and values. To understands this definition better consider the situation when we build the derivative on regular expression r_{i-1}. For this we chop off a character from r_{i-1} to form r_i. This leaves a "hole" in r_i and its corresponding value v_i. To calculate v_{i-1}, we need to locate where that hole is and fill it. We can find this location by comparing r_{i-1} and v_i. For instance, if r_{i-1} is of shape $r_a \cdot r_b$, and v_i is of shape $Left(Seq(v_1, v_2))$, we know immediately that

$$(r_a \cdot r_b)\backslash c = (r_a\backslash c) \cdot r_b \,+\, r_b\backslash c,$$

otherwise if r_a is not nullable,

$$(r_a \cdot r_b)\backslash c = (r_a\backslash c) \cdot r_b,$$

the value v_i should be $Seq(\ldots)$, contradicting the fact that v_i is actually of shape $Left(\ldots)$. Furthermore, since v_i is of shape $Left(\ldots)$ instead of $Right(\ldots)$, we know that the left branch of

$$(r_a \cdot r_b)\backslash c = \underline{(\mathbf{r_a}\backslash \mathbf{c}) \cdot \mathbf{r_b}} \,+\, r_b\backslash c,$$

(underlined) is taken instead of the right one. This means c is chopped off from r_a rather than r_b. We have therefore found out that the hole will be on r_a. So we recursively call $inj\,r_a\,c\,v_a$ to fill that hole in v_a. After injection, the value v_i for $r_i = r_a \cdot r_b$ should be $Seq\,(inj\,r_a\,c\,v_a)\,v_b$. Other clauses can be understood in a similar way.

The following example gives an insight of *inj*'s effect and how Sulzmann and Lu's algorithm works as a whole. Suppose we have a regular expression $((((a+b)+ab)+c)+abc)^*$, and want to match it against the string *abc* (when *abc* is written as a regular expression, the standard way of expressing it is $a \cdot (b \cdot c)$. But we usually omit the parentheses and dots here for better readability. This algorithm returns a POSIX value, which means it will produce the longest matching. Consequently, it matches the string *abc* in one star iteration, using the longest alternative *abc* in the sub-expression (we shall use r to denote this sub-expression for conciseness):

$$((((a+b)+ab)+c)+\underbrace{abc}_{r})$$

Before *inj* is called, our lexer first builds derivative using string *abc* (we simplified some regular expressions like $\mathbf{0}\cdot b$ to $\mathbf{0}$ for conciseness; we also omit parentheses if they are clear from the context):

$$\begin{array}{lll} r^* & \xrightarrow{\backslash a} & r_1 = (\mathbf{1}+\mathbf{0}+\mathbf{1}\cdot b+\mathbf{0}+\mathbf{1}\cdot b\cdot c)\cdot r^* \\ & \xrightarrow{\backslash b} & r_2 = (\mathbf{0}+\mathbf{0}+\mathbf{1}\cdot \mathbf{1}+\mathbf{0}+\mathbf{1}\cdot \mathbf{1}\cdot c)\cdot r^* + (\mathbf{0}+\mathbf{1}+\mathbf{0}+\mathbf{0}+\mathbf{0})\cdot r^* \\ & \xrightarrow{\backslash c} & r_3 = ((\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1})\cdot r^* + (\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}+\mathbf{0})\cdot r^*)+ \\ & & \qquad ((\mathbf{0}+\mathbf{1}+\mathbf{0}+\mathbf{0}+\mathbf{0})\cdot r^* + (\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}+\mathbf{0})\cdot r^*) \end{array}$$

In case r_3 is nullable, we can call *mkeps* to construct a lexical value for how r_3 matched the string *abc*. This function gives the following value v_3:

$$\mathit{Left}(\mathit{Left}(\mathit{Seq}(\mathit{Right}(\mathit{Seq}(\mathit{Empty},\mathit{Seq}(\mathit{Empty},\mathit{Empty}))),\mathit{Stars}[])))$$

The outer $\mathit{Left}(\mathit{Left}(\ldots))$ tells us the leftmost nullable part of r_3(underlined):

$$\begin{array}{ll} & \big(\underline{(\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1})\cdot r^*}\; + \;(\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}+\mathbf{0})\cdot r^*\big) \\ + & \big((\mathbf{0}+\mathbf{1}+\mathbf{0}+\mathbf{0}+\mathbf{0})\cdot r^*\; + \;(\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}+\mathbf{0})\cdot r^*\big) \end{array}$$

Note that the leftmost location of term $(\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1})\cdot r^*$ (which corresponds to the initial sub-match *abc*) allows *mkeps* to pick it up because *mkeps* is defined to always choose the left one when it is nullable. In the case of this example, *abc* is preferred over *a* or *ab*. This $\mathit{Left}(\mathit{Left}(\ldots))$ location is generated by two applications of the splitting clause

$$(r_1\cdot r_2)\backslash c \;\; (\mathit{when}\ r_1\ \mathit{nullable}) \; = (r_1\backslash c)\cdot r_2 \; + \; r_2\backslash c.$$

By this clause, we put $r_1\backslash c\cdot r_2$ at the *front* and $r_2\backslash c$ at the *back*. This allows *mkeps* to always pick up among two matches the one with a longer initial sub-match. Removing the outside $\mathit{Left}(\mathit{Left}(...))$, the inside sub-value

$$\mathit{Seq}(\mathit{Right}(\mathit{Seq}(\mathit{Empty},\mathit{Seq}(\mathit{Empty},\mathit{Empty}))),\mathit{Stars}[])$$

tells us how the empty string [] is matched with $(\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{0}+\mathbf{1}\cdot\mathbf{1}\cdot\mathbf{1})\cdot r^*$. We match [] by a sequence of two nullable regular expressions. The first one is an alternative, we take the rightmost alternative—whose language contains the empty string. The second nullable regular expression is a Kleene star. *Stars* tells us how it generates the nullable regular expression: by 0 iterations to form $\mathbf{1}$. Now *inj* injects characters back and incrementally builds a lexical value based on v_3. Using the value v_3, the character c, and the regular expression r_2, we can recover how r_2 matched the string $[c]$: $\mathit{inj}\, r_2\; c\; v_3$ gives us

$$v_2 = \mathit{Left}(\mathit{Seq}(\mathit{Right}(\mathit{Seq}(\mathit{Empty},\mathit{Seq}(\mathit{Empty},c))),\mathit{Stars}[])),$$

which tells us how r_2 matched $[c]$. After this we inject back the character b, and get

$$v_1 = \mathit{Seq}(\mathit{Right}(\mathit{Seq}(\mathit{Empty},\mathit{Seq}(b,c))),\mathit{Stars}[])$$

for how

$$r_1 = (\mathbf{1}+\mathbf{0}+\mathbf{1}\cdot b+\mathbf{0}+\mathbf{1}\cdot b\cdot c)\cdot r*$$

matched the string *bc* before it split into two substrings. Finally, after injecting character a back to v_1, we get the lexical value tree

$$v_0 = Stars[Right(Seq(a, Seq(b, c)))]$$

for how r matched *abc*. This completes the algorithm.

Readers might have noticed that the lexical value information is actually already available when doing derivatives. For example, immediately after the operation $\backslash a$ we know that if we want to match a string that starts with a, we can either take the initial match to be

1) just a or
2) string *ab* or
3) string *abc*.

In order to differentiate between these choices, we just need to remember their positions—a is on the left, *ab* is in the middle , and *abc* is on the right. Which of these alternatives is chosen later does not affect their relative position because the algorithm does not change this order. If this parsing information can be determined and does not change because of later derivatives, there is no point in traversing this information twice. This leads to an optimisation—if we store the information for lexical values inside the regular expression, update it when we do derivative on them, and collect the information when finished with derivatives and call *mkeps* for deciding which branch is POSIX, we can generate the lexical value in one pass, instead of doing the rest n injections. This leads to Sulzmann and Lu's novel idea of using bitcodes in derivatives.

In the next section, we shall focus on the bitcoded algorithm and the process of simplification of regular expressions. This is needed in order to obtain *fast* versions of the Brzozowski's, and Sulzmann and Lu's algorithms. This is where the PhD-project aims to advance the state-of-the-art.

4 Simplification of Regular Expressions

Using bitcodes to guide parsing is not a novel idea. It was applied to context free grammars and then adapted by Henglein and Nielson for efficient regular expression lexing using DFAs [**?**]. Sulzmann and Lu took this idea of bitcodes a step further by integrating bitcodes into derivatives. The reason why we want to use bitcodes in this project is that we want to introduce more aggressive simplification rules in order to keep the size of derivatives small throughout. This is because 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 *der* 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 simplification 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 [**?**].

Unfortunately, the simplification rules outlined above are not sufficient to prevent a size explosion in all cases. We believe a tighter bound can be achieved that prevents an explosion in *all* cases. Such a tighter 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 [**?**]. He defined the *partial derivatives* of regular expressions as follows:

$$\begin{array}{lcl}
\textit{pder}\; c\; \mathbf{0} & \stackrel{\text{def}}{=} & \emptyset \\
\textit{pder}\; c\; \mathbf{1} & \stackrel{\text{def}}{=} & \emptyset \\
\textit{pder}\; c\; d & \stackrel{\text{def}}{=} & \textit{if}\; c = d\; \{\mathbf{1}\}\; \textit{else}\; \emptyset \\
\textit{pder}\; c\; r_1 + r_2 & \stackrel{\text{def}}{=} & \textit{pder}\; c\; r_1 \cup \textit{pder}\; c\; r_2 \\
\textit{pder}\; c\; r_1 \cdot r_2 & \stackrel{\text{def}}{=} & \textit{if}\; \textit{nullable}\; r_1 \\
 & & \textit{then}\; \{r \cdot r_2 \mid r \in \textit{pder}\; c\; r_1\} \cup \textit{pder}\; c\; r_2 \\
 & & \textit{else}\; \{r \cdot r_2 \mid r \in \textit{pder}\; c\; r_1\} \\
\textit{pder}\; c\; r^* & \stackrel{\text{def}}{=} & \{r' \cdot r^* \mid r' \in \textit{pder}\; c\; r\}
\end{array}$$

A partial derivative of a regular expression r is essentially a set of regular expressions that are either r's children expressions or a concatenation of them. Antimirov has proved a tight bound of the sum of the size of *all* partial derivatives no matter what the string looks like. Roughly speaking the size sum will be at most cubic in the size of the regular expression.

If we want the size of derivatives in Sulzmann and Lu's algorithm to stay below this bound, we would need more aggressive simplifications. Essentially we need to delete useless $\mathbf{0}$s and $\mathbf{1}$s, as well as deleting duplicates whenever possible. For example, the parentheses in $(a+b)\cdot c + bc$ can be opened up to get $a\cdot c + b\cdot c + b\cdot c$, and then simplified to just $a\cdot c + b\cdot c$. Another example is simplifying $(a^* + a) + (a^* + \mathbf{1}) + (a + \mathbf{1})$ to just $a^* + a + \mathbf{1}$. Adding these more aggressive simplification rules helps us to achieve the same size bound as that of the partial derivatives.

In order to implement the idea of "spilling out alternatives" and to make them compatible with the inj-mechanism, we use *bitcodes*. Bits and bitcodes (lists of bits) are just:

$$b ::= S \mid Z \qquad bs ::= [] \mid b : bs$$

The S and Z are arbitrary names for the bits in order to avoid confusion with the regular expressions $\mathbf{0}$ and $\mathbf{1}$. Bitcodes (or bit-lists) can be used to encode values (or incomplete values) in a compact form. This can be straightforwardly seen in the following coding function from values to bitcodes:

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

Here *code* encodes a value into a bitcodes by converting *Left* into Z, *Right* into S, the start point of a non-empty star iteration into S, and the border where a local star terminates into Z. This coding is lossy, as it throws away the information about characters, and also

does not encode the "boundary" between two sequence values. Moreover, with only the bitcode we cannot even tell whether the Ss and Zs are for *Left/Right* or *Stars*. The reason for choosing this compact way of storing information is that the relatively small size of bits can be easily manipulated and "moved around" in a regular expression. In order to recover values, we will need the corresponding regular expression as an extra information. This means the decoding function is defined as:

$$
\begin{array}{lcl}
\textit{decode}'\,bs\,(\mathbf{1}) & \stackrel{\text{def}}{=} & (\textit{Empty}, bs)\\
\textit{decode}'\,bs\,(c) & \stackrel{\text{def}}{=} & (\textit{Char}\,c, bs)\\
\textit{decode}'\,(Z\!::\!bs)\;(r_1 + r_2) & \stackrel{\text{def}}{=} & \textit{let}\,(v, bs_1) = \textit{decode}'\,bs\,r_1\;\textit{in}\;(\textit{Left}\,v, bs_1)\\
\textit{decode}'\,(S\!::\!bs)\;(r_1 + r_2) & \stackrel{\text{def}}{=} & \textit{let}\,(v, bs_1) = \textit{decode}'\,bs\,r_2\;\textit{in}\;(\textit{Right}\,v, bs_1)\\
\textit{decode}'\,bs\;(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \textit{let}\,(v_1, bs_1) = \textit{decode}'\,bs\,r_1\;\textit{in}\\
 & & \textit{let}\,(v_2, bs_2) = \textit{decode}'\,bs_1\,r_2\\
 & & \qquad\textit{in}\;(\textit{Seq}\,v_1\,v_2, bs_2)\\
\textit{decode}'\,(Z\!::\!bs)\,(r^*) & \stackrel{\text{def}}{=} & (\textit{Stars}\,[], bs)\\
\textit{decode}'\,(S\!::\!bs)\,(r^*) & \stackrel{\text{def}}{=} & \textit{let}\,(v, bs_1) = \textit{decode}'\,bs\,r\;\textit{in}\\
 & & \textit{let}\,(\textit{Stars}\,vs, bs_2) = \textit{decode}'\,bs_1\,r^*\\
 & & \qquad\textit{in}\;(\textit{Stars}\,v\!::\!vs, bs_2)\\
 & & \\
\textit{decode}\,bs\,r & \stackrel{\text{def}}{=} & \textit{let}\,(v, bs') = \textit{decode}'\,bs\,r\;\textit{in}\\
 & & \textit{if}\,bs' = []\;\textit{then}\;\textit{Some}\,v\;\textit{else}\;\textit{None}
\end{array}
$$

Sulzmann and Lu's integrated the bitcodes into regular expressions to create annotated regular expressions [**?**]. *Annotated regular expressions* are defined by the following grammar:

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

where bs stands for bitcodes, a for **a**nnotated regular expressions and as for a list of annotated regular expressions. The alternative constructor($\textit{ALTS}$) has been generalized to accept a list of annotated regular expressions rather than just 2. We will show that these bitcodes encode information about the (POSIX) value that should be generated by the Sulzmann and Lu algorithm.

To do lexing using annotated regular expressions, we shall first transform the usual (un-annotated) regular expressions into annotated regular expressions. This operation is called *internalisation* and defined as follows:

$$
\begin{array}{lcl}
(\mathbf{0})^\uparrow & \stackrel{\text{def}}{=} & \textit{ZERO}\\
(\mathbf{1})^\uparrow & \stackrel{\text{def}}{=} & \textit{ONE}\,[]\\
(c)^\uparrow & \stackrel{\text{def}}{=} & \textit{CHAR}\,[]\,c\\
(r_1 + r_2)^\uparrow & \stackrel{\text{def}}{=} & \textit{ALTS}\,[]\,\textit{List}((\textit{fuse}\,[Z]\,r_1^\uparrow),\,(\textit{fuse}\,[S]\,r_2^\uparrow))\\
(r_1 \cdot r_2)^\uparrow & \stackrel{\text{def}}{=} & \textit{SEQ}\,[]\,r_1^\uparrow\,r_2^\uparrow\\
(r^*)^\uparrow & \stackrel{\text{def}}{=} & \textit{STAR}\,[]\,r^\uparrow
\end{array}
$$

We use up arrows here to indicate that the basic un-annotated regular expressions are "lifted up" into something slightly more complex. In the fourth clause, *fuse* is an auxiliary function that helps to attach bits to the front of an annotated regular expression. Its definition is as follows:

$$\begin{array}{lcl}
\textit{fuse}\;bs\,(\textit{ZERO}) & \stackrel{\text{def}}{=} & \textit{ZERO}\\
\textit{fuse}\;bs\,(\textit{ONE}\,bs') & \stackrel{\text{def}}{=} & \textit{ONE}\,(bs\,@\,bs')\\
\textit{fuse}\;bs\,(\textit{CHAR}\,bs'\,c) & \stackrel{\text{def}}{=} & \textit{CHAR}\,(bs\,@\,bs')\,c\\
\textit{fuse}\;bs\,(\textit{ALTS}\,bs'\,as) & \stackrel{\text{def}}{=} & \textit{ALTS}\,(bs\,@\,bs')\,as\\
\textit{fuse}\;bs\,(\textit{SEQ}\,bs'\,a_1\,a_2) & \stackrel{\text{def}}{=} & \textit{SEQ}\,(bs\,@\,bs')\,a_1\,a_2\\
\textit{fuse}\;bs\,(\textit{STAR}\,bs'\,a) & \stackrel{\text{def}}{=} & \textit{STAR}\,(bs\,@\,bs')\,a
\end{array}$$

After internalising the regular expression, we perform successive derivative operations on the annotated regular expressions. This derivative operation is the same as what we had previously for the basic regular expressions, except that we beed to take care of the bitcodes:

$$\begin{array}{lcl}
(\textit{ZERO})\,\backslash c & \stackrel{\text{def}}{=} & \textit{ZERO}\\
(\textit{ONE}\;bs)\,\backslash c & \stackrel{\text{def}}{=} & \textit{ZERO}\\
(\textit{CHAR}\;bs\,d)\,\backslash c & \stackrel{\text{def}}{=} & \textit{if}\;c=d\;\;\textit{then}\;\textit{ONE}\,bs\;\textit{else}\;\textit{ZERO}\\
(\textit{ALTS}\;bs\,as)\,\backslash c & \stackrel{\text{def}}{=} & \textit{ALTS}\;bs\,(as.map(\backslash c))\\
(\textit{SEQ}\;bs\,a_1\,a_2)\,\backslash c & \stackrel{\text{def}}{=} & \textit{if}\;\textit{bnullable}\,a_1\\
 & & \textit{then}\;\textit{ALTS}\,bs\;\textit{List}((\textit{SEQ}\,[]\,(a_1\,\backslash c)\,a_2),\\
 & & \qquad\qquad(\textit{fuse}\,(\textit{bmkeps}\,a_1)\,(a_2\,\backslash c)))\\
 & & \textit{else}\;\textit{SEQ}\,bs\,(a_1\,\backslash c)\,a_2\\
(\textit{STAR}\,bs\,a)\,\backslash c & \stackrel{\text{def}}{=} & \textit{SEQ}\;bs\,(\textit{fuse}\,[Z](r\,\backslash c))\,(\textit{STAR}\,[]\,r)
\end{array}$$

For instance, when we unfold $STAR\ bs\ a$ into a sequence, we need to attach an additional bit Z to the front of $r\backslash c$ to indicate that there is one more star iteration. Also the SEQ clause is more subtle—when a_1 is *bnullable* (here *bnullable* is exactly the same as *nullable*, except that it is for annotated regular expressions, therefore we omit the definition). Assume that *bmkeps* correctly extracts the bitcode for how a_1 matches the string prior to character c (more on this later), then the right branch of $ALTS$, which is $fuse\ bmkeps\ a_1(a_2\backslash c)$ will collapse the regular expression a_1(as it has already been fully matched) and store the parsing information at the head of the regular expression $a_2\backslash c$ by fusing to it. The bitsequence bs, which was initially attached to the head of SEQ, has now been elevated to the top-level of $ALTS$, as this information will be needed whichever way the SEQ is matched—no matter whether c belongs to a_1 or a_2. After building these derivatives and maintaining all the lexing information, we complete the lexing by collecting the bitcodes using a generalised version of the *mkeps* function for annotated regular expressions, called *bmkeps*:

$$\begin{array}{lcl}
\textit{bmkeps}\,(\textit{ONE}\;bs) & \stackrel{\text{def}}{=} & bs\\
\textit{bmkeps}\,(\textit{ALTS}\;bs\,a::as) & \stackrel{\text{def}}{=} & \textit{if}\;\textit{bnullable}\,a\\
 & & \textit{then}\;bs\,@\,\textit{bmkeps}\,a\\
 & & \textit{else}\;bs\,@\,\textit{bmkeps}\,(\textit{ALTS}\;bs\,as)\\
\textit{bmkeps}\,(\textit{SEQ}\;bs\,a_1\,a_2) & \stackrel{\text{def}}{=} & bs\,@\,\textit{bmkeps}\,a_1\,@\,\textit{bmkeps}\,a_2\\
\textit{bmkeps}\,(\textit{STAR}\;bs\,a) & \stackrel{\text{def}}{=} & bs\,@\,[S]
\end{array}$$

This function completes the value information by travelling along the path of the regular expression that corresponds to a POSIX value and collecting all the bitcodes, and using S to

indicate the end of star iterations. If we take the bitcodes produced by *bmkeps* and decode them, we get the value we expect. The corresponding lexing algorithm looks as follows:

$$\begin{array}{lcl} \textit{blexer}\;r\;s & \stackrel{\text{def}}{=} & \textit{let}\;a = (r^\uparrow)\backslash s\;\textit{in} \\ & & \quad \textit{if}\;\textit{bnullable}(a) \\ & & \quad \textit{then}\;\textit{decode}\,(\textit{bmkeps}\,a)\,r \\ & & \quad \textit{else}\;\textit{None} \end{array}$$

In this definition $_\backslash s$ is the generalisation of the derivative operation from characters to strings (just like the derivatives for un-annotated regular expressions).

The main point of the bitcodes 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 partially cover 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 $\mathbf{0}$s and $\mathbf{1}$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 pruned away from the regular expression. A recursive definition of our simplification function that looks somewhat similar to our Scala code is given below:

$$\begin{array}{lcl} \textit{simp}\;(\textit{SEQ}\;bs\,a_1\,a_2) & \stackrel{\text{def}}{=} & (\textit{simp}\;a_1, \textit{simp}\;a_2)\;\textit{match} \\ & & \quad \textit{case}\;(\mathbf{0}, _) \Rightarrow \mathbf{0} \\ & & \quad \textit{case}\;(_, \mathbf{0}) \Rightarrow \mathbf{0} \\ & & \quad \textit{case}\;(\mathbf{1}, a_2') \Rightarrow \textit{fuse}\;bs\;a_2' \\ & & \quad \textit{case}\;(a_1', \mathbf{1}) \Rightarrow \textit{fuse}\;bs\;a_1' \\ & & \quad \textit{case}\;(a_1', a_2') \Rightarrow \textit{SEQ}\;bs\;a_1'\;a_2' \\ \textit{simp}\;(\textit{ALTS}\;bs\,as) & \stackrel{\text{def}}{=} & \textit{distinct}(\textit{flatten}(\textit{map}\;\textit{simp}\;as))\;\textit{match} \\ & & \quad \textit{case}\;[] \Rightarrow \mathbf{0} \\ & & \quad \textit{case}\;a :: [] \Rightarrow \textit{fuse}\;bs\;a \\ & & \quad \textit{case}\;as' \Rightarrow \textit{ALTS}\;bs\;as' \\ \textit{simp}\;a & \stackrel{\text{def}}{=} & a \qquad \textit{otherwise} \end{array}$$

The simplification does a pattern matching on the regular expression. When it detected that the regular expression is an alternative or sequence, it will try to simplify its children regular expressions recursively and then see if one of the children turn into $\mathbf{0}$ or $\mathbf{1}$, which might trigger further simplification at the current level. The most involved part is the *ALTS* clause, where we use two auxiliary functions *flatten* and *distinct* to open up nested *ALTS* and reduce as many duplicates as possible. Function *distinct* keeps the first occurring copy only and remove all later ones when detected duplicates. Function *flatten* opens up nested *ALTS*. Its recursive definition is given below:

$$\begin{array}{lcl} \textit{flatten}\;(\textit{ALTS}\;bs\,as) :: as' & \stackrel{\text{def}}{=} & (\textit{map}\;(\textit{fuse}\;bs)\;as)\;@\;\textit{flatten}\;as' \\ \textit{flatten}\;\textit{ZERO} :: as' & \stackrel{\text{def}}{=} & \textit{flatten}\;as' \\ \textit{flatten}\;a :: as' & \stackrel{\text{def}}{=} & a :: \textit{flatten}\;as' \quad \text{(otherwise)} \end{array}$$

Here *flatten* behaves like the traditional functional programming flatten function, except that it also removes $\mathbf{0}$s. Or in terms of regular expressions, it removes parentheses, for example changing $a + (b + c)$ into $a + b + c$.

Suppose we apply simplification after each derivative step, and view these two operations as an atomic one: $a \backslash_{simp} c \stackrel{\text{def}}{=} simp(a \backslash c)$. Then we can use the previous natural extension from derivative w.r.t. character to derivative w.r.t. string:

$$\begin{array}{lcl} r\backslash_{simp}(c::s) & \stackrel{\text{def}}{=} & (r\backslash_{simp}\, c)\backslash_{simp}\, s \\ r\backslash_{simp}[\,] & \stackrel{\text{def}}{=} & r \end{array}$$

we obtain an optimised version of the algorithm:

$$\begin{array}{lcl} \textit{blexer_simp}\; r\; s & \stackrel{\text{def}}{=} & \textit{let}\; a = (r^\uparrow)\backslash_{simp}\, s\; \textit{in} \\ & & \quad \textit{if}\; \textit{bnullable}(a) \\ & & \quad \textit{then}\; \textit{decode}\, (\textit{bmkeps}\, a)\, r \\ & & \quad \textit{else}\; \textit{None} \end{array}$$

This algorithm keeps the regular expression size small, for example, with this simplification our previous $(a + aa)^*$ example's 8000 nodes will be reduced to just 6 and stays constant, no matter how long the input string is.

5 Current Work

We are currently engaged in two tasks related to this algorithm. The first task is 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 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. 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 lexing 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 tree [our lexical values]. 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 like to settle this correctness claim. It is relatively straightforward to establish that after one simplification step, the part of a nullable derivative that corresponds to a POSIX value remains intact and can still be collected, in other words, we can show that

$$\textit{bmkeps}\; a = \textit{bmkeps}\; \textit{bsimp}\; a\; (\textit{provided}\; a\; \textit{is}\; \textit{bnullable})$$

as this basically comes down to proving actions like removing the additional r in $r + r$ does not delete important POSIX information in a regular expression. The hard part of this proof is to establish that

$$\textit{blexer_simp}(r,\; s) = \textit{blexer}(r,\; s)$$

That is, if we do derivative on regular expression r and then simplify it, and repeat this process until we exhaust the string, we get a regular expression r''(LHS) that provides the POSIX matching information, which is exactly the same as the result r'(RHS of the normal derivative algorithm that only does derivative repeatedly and has no simplification at all. This might seem at first glance very unintuitive, as the r' could be exponentially larger than r'', but can be explained in the following way: we are pruning away the possible matches that are not POSIX. Since there could be exponentially many non-POSIX matchings and only 1 POSIX matching, it is understandable that our r'' can be a lot smaller. we can still provide the same POSIX value if there is one. This is not as straightforward as the previous proposition, as the two regular expressions r' and r'' might have become very different. The crucial point is to find the *POSIX* information of a regular expression and how it is modified, augmented and propagated during simplification in parallel with the regular expression that has not been simplified in the subsequent derivative operations. To aid this, we use the helper function retrieve described by Sulzmann and Lu:

$$
\begin{array}{lcl}
\textit{retrieve}\,(\textit{ONE}\,bs)\;\textit{Empty} & \stackrel{\text{def}}{=} & bs\\
\textit{retrieve}\,(\textit{CHAR}\,bs\,c)\;(\textit{Char}\,d) & \stackrel{\text{def}}{=} & bs\\
\textit{retrieve}\,(\textit{ALTS}\,bs\,a::as)\;(\textit{Left}\,v) & \stackrel{\text{def}}{=} & bs\,@\,\textit{retrieve}\,a\,v\\
\textit{retrieve}\,(\textit{ALTS}\,bs\,a::as)\;(\textit{Right}\,v) & \stackrel{\text{def}}{=} & bs\,@\,\textit{retrieve}\,(\textit{ALTS}\,bs\,as)\,v\\
\textit{retrieve}\,(\textit{SEQ}\,bs\,a_1\,a_2)\;(\textit{Seq}\,v_1\,v_2) & \stackrel{\text{def}}{=} & bs\,@\,\textit{retrieve}\,a_1\,v_1\,@\,\textit{retrieve}\,a_2\,v_2\\
\textit{retrieve}\,(\textit{STAR}\,bs\,a)\;(\textit{Stars}\,[]) & \stackrel{\text{def}}{=} & bs\,@\,[S]\\
\textit{retrieve}\,(\textit{STAR}\,bs\,a)\;(\textit{Stars}\,(v::vs)) & \stackrel{\text{def}}{=} & \\
\multicolumn{3}{r}{bs\,@\,[Z]\,@\,\textit{retrieve}\,a\,v\,@\,\textit{retrieve}\,(\textit{STAR}\,[]\,a)\;(\textit{Stars}\,vs)}
\end{array}
$$

This function assembles the bitcode using information from both the derivative regular expression and the value. Sulzmann and Lu poroposed this function, but did not prove anything about it. Ausaf and Urban used it to connect the bitcoded algorithm to the older algorithm by the following equation:

$$inj\ a\ c\ v = \textit{decode}\ (\textit{retrieve}\ (r^\uparrow)\backslash_{simp}\, c)\ v)$$

whereby $r^\uparrow$ stands for the internalised version of r. Ausaf and Urban also used this fact to prove the correctness of bitcoded algorithm without simplification. Our purpose of using this, however, is to establish

$$\textit{retrieve}\ a\ v \;=\; \textit{retrieve}\ (\textit{simp}\,a)\ v'.$$

The idea is that using v', a simplified version of v that had gone through the same simplification step as $\textit{simp}(a)$, we are able to extract the bitcode that gives the same parsing information as the unsimplified one. However, we noticed that constructing such a v' from v is not so straightforward. The point of this is that we might be able to finally bridge the gap by proving

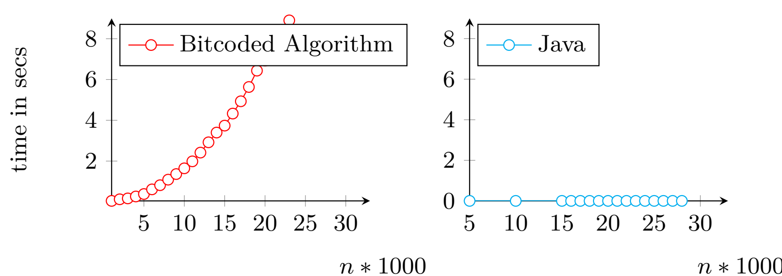
$$\textit{retrieve}\ (r^\uparrow\backslash s)\ v = \;\textit{retrieve}\ (\textit{simp}(r^\uparrow)\backslash s)\ v'$$

and subsequently

$$\textit{retrieve}\ (r^\uparrow\backslash s)\ v \;=\; \textit{retrieve}\ (r^\uparrow\backslash_{simp}\, s)\ v'.$$

The *LHS* of the above equation is the bitcode we want. This would prove that our simplified version of regular expression still contains all the bitcodes needed. The task here is to find a way to compute the correct v'.

The second task is to speed up the more aggressive simplification. Currently it is slower than the original naive simplification by Ausaf and Urban (the naive version as implemented by Ausaf and Urban of course can "explode" in some cases). It is therefore not surprising that the speed is also much slower than regular expression engines in popular programming languages such as Java and Python on most inputs that are linear. For example, just by rewriting the example regular expression in the beginning of this report $(a^*)^*\,b$ into $a^*\,b$ would eliminate the ambiguity in the matching and make the time for matching linear with respect to the input string size. This allows the DFA approach to become blindingly fast, and dwarf the speed of our current implementation. For example, here is a comparison of Java regex engine and our implementation on this example.



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

Java regex engine can match string of thousands of characters in a few milliseconds, whereas our current algorithm gets excruciatingly slow on input of this size. The running time in theory is linear, however it does not appear to be the case in an actual implementation. So it needs to be explored how to make our algorithm faster on all inputs. It could be the recursive calls that are needed to manipulate bits that are causing the slow down. A possible solution is to write recursive functions into tail-recusive form. Another possibility would be to explore again the connection to DFAs to speed up the algorithm on subcalls that are small enough. This is very much work in progress.

6 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. We also plan to implement the bitcoded algorithm in some imperative

language like C to see if the inefficiency of the Scala implementation is language specific. To make this research more comprehensive we also plan to contrast our (faster) version of bitcoded algorithm with the Symbolic Regex Matcher, the RE2, the Rust Regex Engine, and the static analysis approach by implementing them in the same language and then compare their performance.

References

1 V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.

2 F. Ausaf, R. Dyckhoff, and C. Urban. POSIX Lexing with Derivatives of Regular Expressions (Proof Pearl). In *Proc. of the 7th International Conference on Interactive Theorem Proving (ITP)*, volume 9807 of *LNCS*, pages 69–86, 2016.

3 J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

4 J. C. Davis, C. .A. Coghlan, F. Servant, and D. Lee. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 246–256, 2018.

5 N. B. B. Grathwohl, F. Henglein, and U. T. Rasmussen. A Crash-Course in Regular Expression Parsing and Regular Expressions as Types. Technical report, University of Copenhagen, 2014.

6 C. Kuklewicz. Regex Posix. https://wiki.haskell.org/Regex_Posix.

7 Fritz Henglein Lasse Nielsen. Bit-coded regular expression parsing. *LATA*, 2011.

8 S. Okui and T. Suzuki. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. In *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA)*, volume 6482 of *LNCS*, pages 231–240, 2010.

9 Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *arXiv:1405.7058*, 2017.

10 M. Sulzmann and K. Lu. POSIX Regular Expression Parsing with Derivatives. In *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*, volume 8475 of *LNCS*, pages 203–220, 2014.

11 S. Vansummeren. Type Inference for Unique Pattern Matching. *ACM Transactions on Programming Languages and Systems*, 28(3):389–428, 2006.