KING'S COLLEGE LONDON

DOCTORAL THESIS

POSIX Regular Expression Matching and Lexing

Author:
Chengsong TAN

Supervisor:
Dr. Christian URBAN

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Software Systems
Department or Informatics

March 23, 2022

Declaration of Authorship

I, Chengsong TAN, declare that this thesis titled, "POSIX Regular Expression Matching and Lexing" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

KING'S COLLEGE LONDON

Abstract

Chengsong Tan
Department or Informatics

Doctor of Philosophy

POSIX Regular Expression Matching and Lexing

by Chengsong TAN

This work is a combination of functional algorithms and formal methods. Regular expression matching and lexing has been widely-used and well-implemented in software industry.

Theoretical results say that regular expression matching should be linear with respect to the input. Under a certain class of regular expressions and inputs though, practical implementations suffer from non-linear or even exponential running time, allowing a ReDoS (regular expression denial-of-service) attack.

The reason behind is that regex libraries in popular language engines often want to support richer constructs than the most basic regular expressions, and lexing rather than matching is needed for sub-match extraction.

This work aims to address the above vulnerability by the combination of Brzozowski's derivatives and interactive theorem proving. We give an improved version of Sulzmann and Lu's bit-coded algorithm using derivatives, which come with a formal guarantee in terms of correctness and running time as an Isabelle/HOL proof. Then we improve the algorithm with an even stronger version of simplification, and prove a time bound linear to input and cubic to regular expression size using a technique by Antimirov.

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor…

Contents

List of Figures

List of Tables

List of Abbreviations

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or

Physical Constants

Speed of Light	$c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W (J s^{-1})
ω	angular frequency	rad

For/Dedicated to/To my…

Chapter 1

POSIX Lexing With Bit-codes

The main contribution of this thesis is a proven correct lexing algorithm with formalized time bounds. To our best knowledge, there is no lexing libraries using Brzozowski derivatives that have a provable time guarantee, and claims about running time are usually speculative and backed by thin empirical evidence. For example, Sulzmann and Lu had proposed an algorithm in which they claim a linear running time. But that was falsified by our experiments and the running time is actually $\Omega(2^n)$ in the worst case. A similar claim about a theoretical runtime of $O(n^2)$ is made for the Verbatim lexer, which calculates POSIX matches and is based on derivatives. They formalized the correctness of the lexer, but not the complexity. In the performance evaluation section, they simply analyzed the run time of matching a with the string $\underbrace{a \ldots a}_{\text{n a's}}$ and concluded that the algorithm is quadratic in terms of input length. When we tried out their extracted OCaml code with our example $(a + aa)^*$, the time it took to lex only 40 a's was 5 minutes. We therefore believe our results of a proof of performance on general inputs rather than specific examples a novel contribution.

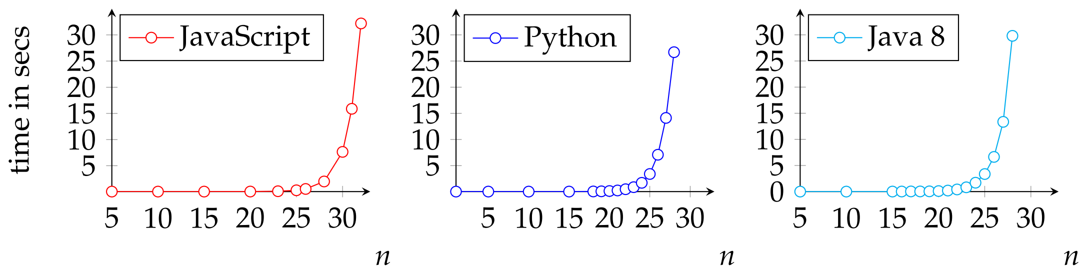
1.1 Introduction To Regexes

Regular expressions are widely used in modern day programming tasks. Be it IDE with syntax hightlighting and auto completion, command line tools like *grep* that facilitates easy processing of text by search and replace, network intrusion detection systems that rejects suspicious traffic, or compiler front-ends–there is always an important phase of the task that involves matching a regular exression with a string. Given its usefulness and ubiquity, one would imagine that modern regular expression matching implementations are mature and fully-studied.

If you go to a popular programming language's regex engine, you can supply it with regex and strings of your own, and get matching/lexing information such as how a sub-part of the regex matches a sub-part of the string. These lexing libraries are on average quite fast. For example, the regex engines some network intrusion detection systems use are able to process megabytes or even gigabytes of network traffic per second.

Why do we need to have our version, if the algorithms work well on average?

Take $(a^*)^*\,b$ 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 $\underbrace{aa..a}_{n}$.

This is clearly exponential behaviour, and is triggered by some relatively simple regex patterns.

The opens up the possibility of a ReDoS (regular expression denial-of-service) attack.

Theoretical results say that regular expression matching should be linear with respect to the input. You could construct an NFA via Thompson construction, and simulate running it. This would be linear. Or you could determinize the NFA into a DFA, and minimize that DFA. Once you have the DFA, the running time is also linear, requiring just one scanning pass of the input.

But modern regex libraries in popular language engines often want to support richer constructs than the most basic regular expressions such as bounded repetitions and back references. These make a DFA construction impossible because of an exponential states explosion. They also want to support lexing rather than just matching for tasks that involves text processing.

Existing regex libraries either pose restrictions on the user input, or does not give linear running time guarantee. For example, the Rust regex engine claims to be linear, but does not support lookarounds and back-references. The GoLang regex library does not support over 1000 repetitions. Java and Python both support back-references, but shows catastrophic backtracking behaviours on inputs without back-references(when the language is still regular). Another thing about the these libraries is that there is no correctness guarantee. In some cases they either fails to generate a lexing result when there is a match, or gives the wrong way of matching.

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 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

It turns out that regex libraries not only suffer from exponential backtracking problems, but also undesired (or even buggy) outputs. xxx commented that most regex libraries are not correctly implementing the POSIX (maximum-munch) rule of regular expression matching. A concrete example would be the regex

```
(((((a*a*)b*)b){20})*)c
```

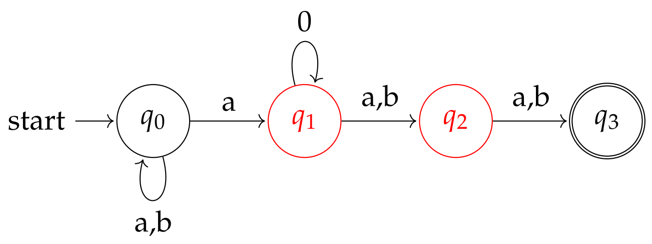
and the string

```
baabaabababaabaaaaaaaaababaa
aababababaaaabaaabaaaaaaabaab
aabababaabababaaaaaaaaababaaaa
babababaaaaaaaaaaaaac
```

This seemingly complex regex simply says "some a's followed by some b's then followed by 1 single b, and this iterates 20 times, finally followed by a c. And a POSIX match would involve the entire string,"eating up" all the b's in it.

This regex would trigger catastrophic backtracking in languages like Python and Java, whereas it gives a correct but uninformative (non-POSIX) match in languages like Go or .NET–The match with only character c.

Backtracking or depth-first search might give us exponential running time, and quite a few tools avoid that by determinising the *NFA* into a *DFA* and minimizes it. For example, *LEX* and *JFLEX* are tools in *C* and *JAVA* that generates *DFA*-based lexers. However, they do not scale well with bounded repetitions. Bounded repetitions, usually written in the form $r^{\{c\}}$ (where c is a constant natural number), denotes a regular expression accepting strings that can be divided into c substrings, and each substring is in r. For the regular expression $(a|b)^*a(a|b)^{\{2\}}$, an *NFA* describing it would look like:



The red states are "counter states" which counts down the number of characters needed in addition to the current string to make a successful match. For example, state q_1 indicates a match that has gone past the $(a|b)$ part of $(a|b)^*a(a|b)^{\{2\}}$, and just consumed the "delimiter" a in the middle, and need to match 2 more iterations of $a|b$ to complete. State q_2 on the other hand, can be viewed as a state after q_1 has consumed 1 character, and just waits for 1 more character to complete. Depending on the actual characters appearing in the input string, the states q_1 and q_2 may or may not be active, independent from each other. A *DFA* for such an *NFA* would contain at least 4 non-equivalent states that cannot be merged, because subset states indicating which of q_1 and q_2 are active are at least four: ϕ, $\{q_1\}$, $\{q_2\}$, $\{q_1, q_2\}$. Generalizing this to regular expressions with larger bounded repetitions number, we have $r^*ar^{\{n\}}$ in general would require at least 2^n states in a *DFA*. This is to represent all different configurations of "countdown" states. For those regexes, tools such as *JFLEX* would generate gigantic *DFA*'s or even give out memory errors.

For this reason, regex libraries that support bounded repetitions often choose to use the *NFA* approach. One can simulate the *NFA* running in two ways: one by keeping track of all active states after consuming a character, and update that set of states iteratively. This is a breadth-first-search of the *NFA*. for a path terminating at an accepting state. Languages like *GO* and *RUST* use this type of *NFA* simulation, and guarantees a linear runtime in terms of input string length. The other way to use *NFA* for matching is to take a single state in a path each time, and backtrack if that path fails. This is a depth-first-search that could end up with exponential run time. The reason behind backtracking algorithms in languages like Java and Python is that they support back-references.

1.1.1 Back References in Regex–Non-Regular part

If we label sub-expressions by parenthesizing them and give them a number by the order their opening parenthesis appear, $\underset{1}{(}\ldots\underset{2}{(}\ldots\underset{3}{(}\ldots\underset{4}{(}\ldots)\ldots)\ldots)\ldots)$ We can use the following syntax to denote that we want a string just matched by a sub-expression to appear at a certain location again exactly: $(.*)\backslash 1$ would match the string like *bobo*, *weewee* and etc.

Back-reference is a construct in the "regex" standard that programmers found quite useful, but not exactly regular any more. In fact, that allows the regex construct to express languages that cannot be contained in context-free languages–they can express things like $a^n a^n a^n$, which is context sensitive.

1.2 Our Solution–Brzozowski Derivatives

Is it possible to have a regex lexing algorithm with proven correctness and time complexity, which allows easy extensions to constructs like bounded repetitions, negation, lookarounds, and even back-references?

We propose Brzozowski's derivatives as a solution to this problem.

1.3 Preliminaries about Lexing Using Brzozowski derivatives

In the last fifteen or so years, Brzozowski's derivatives of regular expressions have sparked quite a bit of interest in the functional programming and theorem prover communities. The beauty of Brzozowski's derivatives (Brzozowski, 1964) is that they are neatly expressible in any functional language, and easily definable and reasoned about in theorem provers—the definitions just consist of inductive datatypes and simple recursive functions.

Suppose we have an alphabet Σ, the strings whose characters are from Σ can be expressed as Σ^*.

We use patterns to define a set of strings concisely. Regular expressions are one of such patterns systems: The basic regular expressions are defined inductively 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 language or set of strings defined by regular expressions are defined as

$$\begin{array}{lcl} L\, r_1 + r_2 & \stackrel{\text{def}}{=} & L\, r_1 \cup L\, r_2 \\ L\, r_1 \cdot r_2 & \stackrel{\text{def}}{=} & L\, r_1 \cap L\, r_2 \end{array}$$

Which are also called the "language interpretation".

The Brzozowski derivative w.r.t character c is an operation on the regex, where the operation transforms the regex to a new one containing strings without the head character c.

Formally, we define first such a transformation on any string set, which we call semantic derivative:

$$Der\ c\ StringSet = \{s \mid c :: s \in StringSet\}$$

Mathematically, it can be expressed as the

If the *StringSet* happen to have some structure, for example, if it is regular, then we have that it

The the derivative of regular expression, denoted as $r\backslash c$, is a function that takes parameters r and c, and returns another regular expression r', which is computed by the following recursive function:

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

The *nullable* function tests whether the empty string "" is in the language of r:

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

The empty set does not contain any string and therefore not the empty string, the empty string regular expression contains the empty string by definition, the character regular expression is the singleton that contains character only, and therefore does not contain the empty string, the alternative regular expression(or "or" expression) might have one of its children regular expressions being nullable and any one of its children being nullable would suffice. The sequence regular expression would require both children to have the empty string to compose an empty string and the Kleene star operation naturally introduced the empty string.

We can give the meaning of regular expressions derivatives by language interpretation:

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

The function derivative, written $\backslash c$, defines how a regular expression evolves into a new regular expression after all the string it contains is chopped off a certain head character c. The most involved cases are the sequence and star case. The sequence case says that if the first regular expression contains an empty string then second component of the sequence might be chosen as the target regular expression to be chopped off its head character. The star regular expression unwraps the iteration of regular expression and attack the star regular expression to its back again to make sure there are 0 or more iterations following this unfolded iteration.

The main property of the derivative operation that enables us to reason about the correctness of an algorithm using derivatives is

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

We can generalise the derivative operation shown above for single characters to strings as follows:

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

and then define Brzozowski's regular-expression matching algorithm as:

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

Assuming the a string is given as a sequence of characters, say $c_0c_1..c_n$, this algorithm presented graphically is as follows:

$$r_0 \xrightarrow{\backslash c_0} r_1 \xrightarrow{\backslash c_1} r_2 \dashrightarrow r_n \xrightarrow{\textit{nullable}?} \text{YES/NO} \qquad (1.1)$$

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 $s \in L(r_0)$).

Beautiful and simple definition.

If we implement the above algorithm naively, however, the algorithm can be 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 $\backslash$ and *nullable* need to traverse such trees and consequently the bigger the size of the derivative the slower the algorithm.

Brzozowski was quick in finding that during this process a lot useless **1**s and **0**s are generated and therefore not optimal. He also introduced some "similarity rules" such as $P + (Q + R) = (P + Q) + R$ to merge syntactically different but language-equivalent sub-regexes to further decrease the size of the intermediate regexes.

More simplifications are possible, such as deleting duplicates and opening up nested alternatives to trigger even more simplifications. And suppose we apply simplification after each derivative step, and compose these two operations together as an atomic one: $a\backslash_{simp}\, c \stackrel{\text{def}}{=} simp(a\backslash c)$. Then we can build a matcher without having cumbersome regular expressions.

If we want the size of derivatives in the algorithm to stay even lower, we would need more aggressive simplifications. Essentially we need to delete useless **0**s and **1**s, as well as deleting duplicates whenever possible. For example, the parentheses in $(a + b) \cdot c + b \cdot c$ 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 help us to achieve a very tight size bound, namely, the same size bound as that of the *partial derivatives*.

Building derivatives and then simplify them. So far so good. But what if we want to do lexing instead of just a YES/NO answer? This requires us to go back again to the world without simplification first for a moment. Sulzmann and Lu **Sulzmann2014** first came up with a nice and elegant(arguably as beautiful as the original derivatives definition) solution for this.

Values and the Lexing Algorithm by Sulzmann and Lu

They first defined the datatypes for storing the lexing information 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	$::=$	
	$\mid$	$\mathbf{1}$			$Empty$
	$\mid$	c		$\mid$	$Char(c)$
	$\mid$	$r_1 \cdot r_2$		$\mid$	$Seq\, v_1\, v_2$
	$\mid$	$r_1 + r_2$		$\mid$	$Left(v)$
				$\mid$	$Right(v)$
	$\mid$	r^*		$\mid$	$Stars\,[v_1, \ldots\, v_n]$

One regular expression can have multiple lexical values. For example for the regular expression $(a + b)^*$, it has a infinite list of values corresponding to it: $Stars\,[]$, $Stars\,[Left(Char(a))]$, $Stars\,[Right(Char(b))]$, $Stars\,[Left(Char(a), Right(Char(b))]$, ..., and vice versa. Even for the regular expression matching a certain string, there could still be more than one value corresponding to it. Take the example where $r = (a^* \cdot a^*)^*$ and the string $s = \underbrace{aa\ldots a}_{\text{n } as}$. The number of different ways of matching without allowing any value under a star to be flattened to an empty string can be given by the following formula:

$$C_n = (n + 1) + nC_1 + \ldots + 2C_{n-1}$$

and a closed form formula can be calculated to be

$$C_n = \frac{(2 + \sqrt{2})^n - (2 - \sqrt{2})^n}{4\sqrt{2}} \tag{1.2}$$

which is clearly in exponential order. A lexer aimed at getting all the possible values has an exponential worst case runtime. Therefore it is impractical to try to generate all possible matches in a run. In practice, we are usually interested about POSIX values, which by intuition always match the leftmost regular expression when there is a choice and always match a sub part as much as possible before proceeding to the next token. For example, the above example has the POSIX value $Stars\,[Seq(Stars\,[\underbrace{Char(a),\ldots,Char(a)}_{\text{n iterations}}], Stars\,[])]$. The output of an algorithm we want would be a POSIX matching encoded as a value. 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 (1.1)). 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:

$$\begin{tikzcd}
r_0 \arrow[r, "\backslash c_0"] \arrow[d] & r_1 \arrow[r, "\backslash c_1"] \arrow[d] & r_2 \arrow[r, dashed] \arrow[d] & r_n \arrow[d, dashed, "mkeps"] \\
v_0 & v_1 \arrow[l, "inj_{r_0} c_0"'] & v_2 \arrow[l, "inj_{r_1} c_1"'] & v_n \arrow[l, dashed]
\end{tikzcd} \tag{1.3}$$

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 a POSIX lexical value for how the empty string has been matched by the (nullable) regular expression r_n. This function is defined as

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

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. The POSIX value is maintained throught out the process. 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. The clauses basically do one thing–identifying the "holes" on value to inject the character back into. For instance, in the last clause for injecting back to a value that would turn into a new star value that corresponds to a star, we know it must be a sequence value. And we know that the first value of that sequence corresponds to the child regex of the star with the first character being chopped off–an iteration of the star that had just been unfolded. This value is followed by the already matched star iterations we collected before. So we inject the character back to the first value and form a new value with this new iteration being added to the previous list of iterations, all under the *Stars* top level.

We have mentioned before that derivatives without simplification can get clumsy, and this is true for values as well–they reflect the regular expressions size by definition.

One can introduce simplification on the regex and values, but have to be careful in not breaking the correctness as the injection function heavily relies on the structure of the regexes and values being correct and match each other. It can be achieved by recording some extra rectification functions during the derivatives step, and applying these rectifications in each run during the injection phase. And we can prove that the POSIX value of how regular expressions match strings will not be affected—although is much harder to establish. Some initial results in this regard have been obtained in **AusafDyckhoffUrban2016**.

We want to get rid of complex and fragile rectification of values. Can we not create those intermediate values $v_1, \ldots v_n$, and get the lexing information that should be already there while doing derivatives in one pass, without a second phase of injection? In the meantime, can we make sure that simplifications are easily handled without breaking the correctness of the algorithm?

Sulzmann and Lu solved this problem by introducing additional informtaion to the regular expressions called *bitcodes*.

Bit-coded Algorithm

Bits and bitcodes (lists of bits) are defined as:

$$b ::= 1 \mid 0 \qquad bs ::= [] \mid b :: bs$$

The 1 and 0 are not in bold in order to avoid confusion with the regular expressions **0** and **1**. Bitcodes (or bit-lists) can be used to encode values (or potentially incomplete values) in a compact form. This can be straightforwardly seen in the following coding function from values to bitcodes:

$$\begin{array}{lcl}
code(Empty) & \stackrel{\text{def}}{=} & [] \\
code(Char\, c) & \stackrel{\text{def}}{=} & [] \\
code(Left\, v) & \stackrel{\text{def}}{=} & 0 :: code(v) \\
code(Right\, v) & \stackrel{\text{def}}{=} & 1 :: code(v) \\
code(Seq\, v_1\, v_2) & \stackrel{\text{def}}{=} & code(v_1)\, @\, code(v_2) \\
code(Stars\, []) & \stackrel{\text{def}}{=} & [0] \\
code(Stars\, (v :: vs)) & \stackrel{\text{def}}{=} & 1 :: code(v)\ @\ code(Stars\, vs)
\end{array}$$

Here *code* encodes a value into a bitcodes by converting *Left* into 0, *Right* into 1, and marks the start of a non-empty star iteration by 1. The border where a local star terminates is marked by 0. 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 1s and 0s 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:

$$\begin{array}{ll}
decode'\ bs\ (\mathbf{1}) & \stackrel{\text{def}}{=} (Empty, bs) \\
decode'\ bs\ (c) & \stackrel{\text{def}}{=} (Char\, c, bs) \\
decode'\ (0 :: bs)\ (r_1 + r_2) & \stackrel{\text{def}}{=} let\ (v, bs_1) = decode'\ bs\, r_1\ in\ (Left\, v, bs_1) \\
decode'\ (1 :: bs)\ (r_1 + r_2) & \stackrel{\text{def}}{=} let\ (v, bs_1) = decode'\ bs\, r_2\ in\ (Right\, v, bs_1) \\
decode'\ bs\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} let\ (v_1, bs_1) = decode'\ bs\, r_1\ in \\
& \quad let\ (v_2, bs_2) = decode'\ bs_1\, r_2 \\
& \qquad\qquad in\ (Seq\, v_1\, v_2, bs_2) \\
decode'\ (0 :: bs)\ (r^*) & \stackrel{\text{def}}{=} (Stars\, [], bs) \\
decode'\ (1 :: bs)\ (r^*) & \stackrel{\text{def}}{=} let\ (v, bs_1) = decode'\ bs\, r\ in \\
& \quad let\ (Stars\, vs, bs_2) = decode'\ bs_1\, r^* \\
& \qquad\qquad in\ (Stars\, v :: vs, bs_2) \\
& \\
decode\, bs\, r & \stackrel{\text{def}}{=} let\ (v, bs') = decode'\ bs\, r\ in \\
& \quad if\ bs' = []\ then\ Some\, v\ else\ None
\end{array}$$

Sulzmann and Lu's integrated the bitcodes into regular expressions to create annotated regular expressions **Sulzmann2014**. *Annotated regular expressions* are defined by the following grammar:

$$\begin{array}{lcl}
a & ::= & \mathbf{0} \\
& | & _{bs}\mathbf{1} \\
& | & _{bs}\mathbf{c} \\
& | & _{bs}\sum as \\
& | & _{bs}a_1 \cdot a_2 \\
& | & _{bs}a^*
\end{array}$$

where *bs* stands for bitcodes, *a* for **a**nnotated regular expressions and *as* for a list of annotated regular expressions. The alternative constructor($\sum$) 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:

$$
\begin{array}{lcl}
(\mathbf{0})^\uparrow & \stackrel{\text{def}}{=} & \mathbf{0} \\
(\mathbf{1})^\uparrow & \stackrel{\text{def}}{=} & _{[]}\mathbf{1} \\
(c)^\uparrow & \stackrel{\text{def}}{=} & _{[]}\mathbf{c} \\
(r_1 + r_2)^\uparrow & \stackrel{\text{def}}{=} & _{[]}\sum[\textit{fuse}\,[0]\,r_1^\uparrow, \textit{fuse}\,[1]\,r_2^\uparrow] \\
(r_1 \cdot r_2)^\uparrow & \stackrel{\text{def}}{=} & _{[]}r_1^\uparrow \cdot r_2^\uparrow \\
(r^*)^\uparrow & \stackrel{\text{def}}{=} & _{[]}(r^\uparrow)^*
\end{array}
$$

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:

$$
\begin{array}{lcl}
\textit{fuse}\;bs\;\mathbf{0} & \stackrel{\text{def}}{=} & \mathbf{0} \\
\textit{fuse}\;bs\;_{bs'}\mathbf{1} & \stackrel{\text{def}}{=} & _{bs@bs'}\mathbf{1} \\
\textit{fuse}\;bs\;_{bs'}\mathbf{c} & \stackrel{\text{def}}{=} & _{bs@bs'}\mathbf{c} \\
\textit{fuse}\;bs\;_{bs'}\sum as & \stackrel{\text{def}}{=} & _{bs@bs'}\sum as \\
\textit{fuse}\;bs\;_{bs'}a_1 \cdot a_2 & \stackrel{\text{def}}{=} & _{bs@bs'}a_1 \cdot a_2 \\
\textit{fuse}\;bs\;_{bs'}a^* & \stackrel{\text{def}}{=} & _{bs@bs'}a^*
\end{array}
$$

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:

$$
\begin{array}{lcl}
(\mathbf{0})\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
(_{bs}\mathbf{1})\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
(_{bs}\mathbf{d})\backslash c & \stackrel{\text{def}}{=} & \textit{if}\; c = d \;\; \textit{then}\; _{bs}\mathbf{1}\; \textit{else}\; \mathbf{0} \\
(_{bs}\sum\; as)\backslash c & \stackrel{\text{def}}{=} & _{bs}\sum\,(as.map(\backslash c)) \\
(_{bs}\; a_1 \cdot a_2)\backslash c & \stackrel{\text{def}}{=} & \textit{if}\; \textit{bnullable}\, a_1 \\
 & & \textit{then}\; _{bs}\sum\,[(_{[]}\,(a_1\backslash c)\cdot\, a_2), \\
 & & \qquad\qquad (\textit{fuse}\,(\textit{bmkeps}\, a_1)\,(a_2\backslash c))] \\
 & & \textit{else}\; _{bs}\,(a_1\backslash c)\cdot a_2 \\
(_{bs}a^*)\backslash c & \stackrel{\text{def}}{=} & _{bs}(\textit{fuse}\,[0]\; r\backslash c)\cdot(_{[]}r^*))
\end{array}
$$

For instance, when we do derivative of $_{bs}a^*$ with respect to c, we need to unfold it into a sequence, and attach an additional bit 0 to the front of $r\backslash c$ to indicate that there is one more star iteration. Also the sequence 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 alternative, 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 first element of the sequence $a_1 \cdot a_2$, has now been elevated to the top-level of $\sum$, as this information will be needed whichever way the sequence 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*:

$$
\begin{array}{lcl}
\mathit{bmkeps}\;(_{bs}\mathbf{1}) & \stackrel{\text{def}}{=} & \mathit{bs}\\
\mathit{bmkeps}\;(_{bs}\sum a :: \mathit{as}) & \stackrel{\text{def}}{=} & \mathit{if}\;\mathit{bnullable}\;a\\
 & & \mathit{then}\;\mathit{bs}\,@\,\mathit{bmkeps}\;a\\
 & & \mathit{else}\;\mathit{bs}\,@\,\mathit{bmkeps}\;(_{bs}\sum \mathit{as})\\
\mathit{bmkeps}\;(_{bs}a_1 \cdot a_2) & \stackrel{\text{def}}{=} & \mathit{bs}\,@\,\mathit{bmkeps}\;a_1\,@\,\mathit{bmkeps}\;a_2\\
\mathit{bmkeps}\;(_{bs}a^*) & \stackrel{\text{def}}{=} & \mathit{bs}\,@\,[0]
\end{array}
$$

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:

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

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).

Remember tha one of the important reasons we introduced bitcodes is that they can make simplification more structured and therefore guaranteeing the correctness.

Our Simplification Rules

In this section we introduce aggressive (in terms of size) simplification rules on annotated regular expressions in order to keep derivatives small. Such simplifications are promising as we have generated test data that show that a good tight 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 $\sum$ 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 pruned away from the regular expression. A recursive definition of our simplification function that looks somewhat similar to our Scala code is given below:

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

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 **0** or **1**, which might trigger further simplification at the current level. The most involved part is the $\sum$ clause, where we use two auxiliary functions *flatten* and *distinct* to open up nested alternatives 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 $\sum$s. Its recursive definition is given below:

$$\begin{array}{lcl}
\textit{flatten}\;({}_{bs}\sum as) :: as' & \stackrel{\text{def}}{=} & (\textit{map}\;(\textit{fuse}\; bs)\; as) \mathbin{@} \textit{flatten}\; as' \\
\textit{flatten}\; \mathbf{0} :: as' & \stackrel{\text{def}}{=} & \textit{flatten}\; as' \\
\textit{flatten}\; a :: as' & \stackrel{\text{def}}{=} & a :: \textit{flatten}\; as' \quad \text{(otherwise)}
\end{array}$$

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

Having defined the *simp* function, we can use the previous notation of natural extension from derivative w.r.t. character to derivative w.r.t. string:

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

to 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.

Derivatives give a simple solution to the problem of matching a string s with a regular expression r: if the derivative of r w.r.t. (in succession) all the characters of the string matches the empty string, then r matches s (and *vice versa*).

However, there are two difficulties with derivative-based matchers: First, Brzozowski's original matcher only generates a yes/no answer for whether a regular expression matches a string or not. This is too little information in the context of lexing where separate tokens must be identified and also classified (for example as keywords or identifiers). Sulzmann and Lu **Sulzmann2014** overcome this difficulty by cleverly extending Brzozowski's matching algorithm. Their extended version generates additional information on *how* a regular expression matches a string following the POSIX rules for regular expression matching. They achieve this by adding a second "phase" to Brzozowski's algorithm involving an injection function. In our own earlier work we provided the formal specification of what POSIX matching means and proved in Isabelle/HOL the correctness of Sulzmann and Lu's extended algorithm accordingly **AusafDyckhoffUrban2016**.

The second difficulty is that Brzozowski's derivatives can grow to arbitrarily big sizes. For example if we start with the regular expression $(a+aa)^*$ and take successive derivatives according to the character a, we end up with a sequence of ever-growing derivatives like

$$\begin{array}{lll}
(a+aa)^* & \xrightarrow{\_\backslash a} & (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* \\
 & \xrightarrow{\_\backslash a} & (\mathbf{0}+\mathbf{0}a+\mathbf{1})\cdot(a+aa)^* \;+\; (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* \\
 & \xrightarrow{\_\backslash a} & (\mathbf{0}+\mathbf{0}a+\mathbf{0})\cdot(a+aa)^* + (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* + \\
 & & \quad (\mathbf{0}+\mathbf{0}a+\mathbf{1})\cdot(a+aa)^* + (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^* \\
 & \xrightarrow{\_\backslash a} & \ldots \qquad\qquad \text{(regular expressions of sizes 98, 169, 283, 468, 767, \ldots)}
\end{array}$$

where after around 35 steps we run out of memory on a typical computer (we shall define shortly the precise details of our regular expressions and the derivative operation). Clearly, the notation involving **0**s and **1**s already suggests simplification rules that can be applied to regular regular expressions, for example $\mathbf{0}\,r \Rightarrow \mathbf{0}$, $\mathbf{1}\,r \Rightarrow r$, $\mathbf{0}+r \Rightarrow r$ and $r+r \Rightarrow r$. While such simple-minded simplifications have been proved in our earlier work to preserve the correctness of Sulzmann and Lu's algorithm **AusafDyckhoffUrban2016**, they unfortunately do *not* help with limiting the growth of the derivatives shown above: the growth is slowed, but the derivatives can still grow rather quickly beyond any finite bound.

Sulzmann and Lu overcome this "growth problem" in a second algorithm **Sulzmann2014** where they introduce bitcoded regular expressions. In this version, POSIX values are represented as bitsequences and such sequences are incrementally generated when derivatives are calculated. The compact representation of bitsequences and regular expressions allows them to define a more "aggressive" simplification method that keeps the size of the derivatives finite no matter what the length of the string is. They make some informal claims about the correctness and linear behaviour of this version, but do not provide any supporting proof arguments, not even "pencil-and-paper" arguments. They write about their bitcoded *incremental parsing method* (that is the algorithm to be formalised in this paper):

> *"Correctness Claim: We further claim that the incremental parsing method [..] in combination with the simplification steps [..] yields POSIX parse trees. We have tested this claim extensively [..] but yet have to work out all proof details."* **Sulzmann2014**

1.4 Backgound

Theoretical results say that regular expression matching should be linear with respect to the input. Under a certain class of regular expressions and inputs though, practical implementations suffer from non-linear or even exponential running time, allowing a ReDoS (regular expression denial-of-service) attack.

1.5 Engineering and Academic Approaches to Deal with Catastrophic Backtracking

The reason behind is that regex libraries in popular language engines often want to support richer constructs than the most basic regular expressions, and lexing rather than matching is needed for sub-match extraction.

There is also static analysis work on regular expression that have potential expoential behavious. Rathnayake and Thielecke (**Rathnayake2014StaticAF**) proposed an algorithm that detects regular expressions triggering exponential behavious on backtracking matchers. People also developed static analysis methods for generating non-linear polynomial worst-time estimates for regexes, attack string that exploit the worst-time scenario, and "attack automata" that generates attack strings. For a comprehensive analysis, please refer to Weideman's thesis (**Weideman2017Static**).

1.5.1 DFA Approach

Exponential states.

1.5.2 NFA Approach

Backtracking.

1.6 Our Approach

In the last fifteen or so years, Brzozowski's derivatives of regular expressions have sparked quite a bit of interest in the functional programming and theorem prover communities. The beauty of Brzozowski's derivatives (Brzozowski, 1964) is that they are neatly expressible in any functional language, and easily definable and reasoned about in theorem provers—the definitions just consist of inductive datatypes and simple recursive functions. Derivatives of a regular expression, written $r\backslash c$, give a simple solution to the problem of matching a string s with a regular expression r: if the derivative of r w.r.t. (in succession) all the characters of the string matches the empty string, then r matches s (and *vice versa*).

This work aims to address the above vulnerability by the combination of Brzozowski's derivatives and interactive theorem proving. We give an improved version of Sulzmann and Lu's bit-coded algorithm using derivatives, which come with a formal guarantee in terms of correctness and running time as an Isabelle/HOL proof. Then we improve the algorithm with an even stronger version of simplification, and prove a time bound linear to input and cubic to regular expression size using a technique by Antimirov.

1.6.1 Existing Work

We are aware of a mechanised correctness proof of Brzozowski's derivative-based matcher in HOL4 by Owens and Slind (Owens and Slind, 2008). Another one in Isabelle/HOL is part of the work by Krauss and Nipkow (Krauss and Nipkow, 2012).

And another one in Coq is given by Coquand and Siles (Coquand and Siles, 2011). Also Ribeiro and Du Bois give one in Agda (Ribeiro and Bois, 2017).

1.7 What this Template Includes

1.7.1 Folders

This template comes as a single zip file that expands out to several files and folders. The folder names are mostly self-explanatory:

Appendices – this is the folder where you put the appendices. Each appendix should go into its own separate `.tex` file. An example and template are included in the directory.

Chapters – this is the folder where you put the thesis chapters. A thesis usually has about six chapters, though there is no hard rule on this. Each chapter should go in its own separate `.tex` file and they can be split as:

- Chapter 1: Introduction to the thesis topic
- Chapter 2: Background information and theory
- Chapter 3: (Laboratory) experimental setup
- Chapter 4: Details of experiment 1
- Chapter 5: Details of experiment 2
- Chapter 6: Discussion of the experimental results
- Chapter 7: Conclusion and future directions

This chapter layout is specialised for the experimental sciences, your discipline may be different.

Figures – this folder contains all figures for the thesis. These are the final images that will go into the thesis document.

1.7.2 Files

Included are also several files, most of them are plain text and you can see their contents in a text editor. After initial compilation, you will see that more auxiliary files are created by LaTeX or BibTeX and which you don't need to delete or worry about:

example.bib – this is an important file that contains all the bibliographic information and references that you will be citing in the thesis for use with BibTeX. You can write it manually, but there are reference manager programs available that will create and manage it for you. Bibliographies in LaTeX are a large subject and you may need to read about BibTeX before starting with this. Many modern reference managers will allow you to export your references in BibTeX format which greatly eases the amount of work you have to do.

MastersDoctoralThesis.cls – this is an important file. It is the class file that tells LaTeX how to format the thesis.

main.pdf – this is your beautifully typeset thesis (in the PDF file format) created by LaTeX. It is supplied in the PDF with the template and after you compile the template you should get an identical version.

main.tex – this is an important file. This is the file that you tell LaTeX to compile to produce your thesis as a PDF file. It contains the framework and constructs that tell LaTeX how to layout the thesis. It is heavily commented so you can read exactly what each line of code does and why it is there. After you put your own information into the *THESIS INFORMATION* block – you have now started your thesis!

Files that are *not* included, but are created by LaTeX as auxiliary files include:

main.aux – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main `.tex` file.

main.bbl – this is an auxiliary file generated by BibTeX, if it is deleted, BibTeX simply regenerates it when you run the `main.aux` file. Whereas the `.bib` file contains all the references you have, this `.bbl` file contains the references you have actually cited in the thesis and is used to build the bibliography section of the thesis.

main.blg – this is an auxiliary file generated by BibTeX, if it is deleted BibTeX simply regenerates it when you run the main `.aux` file.

main.lof – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main `.tex` file. It tells LaTeX how to build the *List of Figures* section.

main.log – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main `.tex` file. It contains messages from LaTeX, if you receive errors and warnings from LaTeX, they will be in this `.log` file.

main.lot – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main `.tex` file. It tells LaTeX how to build the *List of Tables* section.

main.out – this is an auxiliary file generated by LaTeX, if it is deleted LaTeX simply regenerates it when you run the main `.tex` file.

So from this long list, only the files with the `.bib`, `.cls` and `.tex` extensions are the most important ones. The other auxiliary files can be ignored or deleted as LaTeX and BibTeX will regenerate them.

1.8 Filling in Your Information in the `main.tex` File

You will need to personalise the thesis template and make it your own by filling in your own information. This is done by editing the `main.tex` file in a text editor or your favourite LaTeX environment.

Open the file and scroll down to the third large block titled *THESIS INFORMATION* where you can see the entries for *University Name*, *Department Name*, etc . . .

Fill out the information about yourself, your group and institution. You can also insert web links, if you do, make sure you use the full URL, including the `http://` for this. If you don't want these to be linked, simply remove the `\href{url}{name}` and only leave the name.

When you have done this, save the file and recompile `main.tex`. All the information you filled in should now be in the PDF, complete with web links. You can now begin your thesis proper!

1.9 The `main.tex` File Explained

The `main.tex` file contains the structure of the thesis. There are plenty of written comments that explain what pages, sections and formatting the LaTeX code is creating. Each major document element is divided into commented blocks with titles in all capitals to make it obvious what the following bit of code is doing. Initially there seems to be a lot of LaTeX code, but this is all formatting, and it has all been taken care of so you don't have to do it.

Begin by checking that your information on the title page is correct. For the thesis declaration, your institution may insist on something different than the text given. If this is the case, just replace what you see with what is required in the *DECLARATION PAGE* block.

Then comes a page which contains a funny quote. You can put your own, or quote your favourite scientist, author, person, and so on. Make sure to put the name of the person who you took the quote from.

Following this is the abstract page which summarises your work in a condensed way and can almost be used as a standalone document to describe what you have done. The text you write will cause the heading to move up so don't worry about running out of space.

Next come the acknowledgements. On this page, write about all the people who you wish to thank (not forgetting parents, partners and your advisor/supervisor).

The contents pages, list of figures and tables are all taken care of for you and do not need to be manually created or edited. The next set of pages are more likely to be optional and can be deleted since they are for a more technical thesis: insert a list of abbreviations you have used in the thesis, then a list of the physical constants and numbers you refer to and finally, a list of mathematical symbols used in any formulae. Making the effort to fill these tables means the reader has a one-stop place to refer to instead of searching the internet and references to try and find out what you meant by certain abbreviations or symbols.

The list of symbols is split into the Roman and Greek alphabets. Whereas the abbreviations and symbols ought to be listed in alphabetical order (and this is *not* done automatically for you) the list of physical constants should be grouped into similar themes.

The next page contains a one line dedication. Who will you dedicate your thesis to?

Finally, there is the block where the chapters are included. Uncomment the lines (delete the `%` character) as you write the chapters. Each chapter should be written in its own file and put into the *Chapters* folder and named `Chapter1`, `Chapter2`, etc…Similarly for the appendices, uncomment the lines as you need them. Each appendix should go into its own file and placed in the *Appendices* folder.

After the preamble, chapters and appendices finally comes the bibliography. The bibliography style (called `authoryear`) is used for the bibliography and is a fully featured style that will even include links to where the referenced paper can be found online. Do not underestimate how grateful your reader will be to find that a reference to a paper is just a click away. Of course, this relies on you putting the URL information into the BibTeX file in the first place.

1.10 Thesis Features and Conventions

To get the best out of this template, there are a few conventions that you may want to follow.

One of the most important (and most difficult) things to keep track of in such a long document as a thesis is consistency. Using certain conventions and ways of doing things (such as using a Todo list) makes the job easier. Of course, all of these are optional and you can adopt your own method.

1.10.1 Printing Format

This thesis template is designed for double sided printing (i.e. content on the front and back of pages) as most theses are printed and bound this way. Switching to one sided printing is as simple as uncommenting the *`oneside`* option of the `documentclass` command at the top of the `main.tex` file. You may then wish to adjust the margins to suit specifications from your institution.

The headers for the pages contain the page number on the outer side (so it is easy to flick through to the page you want) and the chapter name on the inner side.

The text is set to 11 point by default with single line spacing, again, you can tune the text size and spacing should you want or need to using the options at the very start of `main.tex`. The spacing can be changed similarly by replacing the *`singlespacing`* with *`onehalfspacing`* or *`doublespacing`*.

1.10.2 Using US Letter Paper

The paper size used in the template is A4, which is the standard size in Europe. If you are using this thesis template elsewhere and particularly in the United States, then you may have to change the A4 paper size to the US Letter size. This can be done in the margins settings section in `main.tex`.

Due to the differences in the paper size, the resulting margins may be different to what you like or require (as it is common for institutions to dictate certain margin sizes). If this is the case, then the margin sizes can be tweaked by modifying the values in the same block as where you set the paper size. Now your document should be set up for US Letter paper size with suitable margins.

1.10.3 References

The `biblatex` package is used to format the bibliography and inserts references such as this one (**Reference1**). The options used in the `main.tex` file mean that the in-text citations of references are formatted with the author(s) listed with the date of the publication. Multiple references are separated by semicolons (e.g. (**Reference2**; **Reference1**)) and references with more than three authors only show the first author with *et al.* indicating there are more authors (e.g. (**Reference3**)). This is done automatically for you. To see how you use references, have a look at the `Chapter1.tex` source file. Many reference managers allow you to simply drag the reference into the document as you type.

Scientific references should come *before* the punctuation mark if there is one (such as a comma or period). The same goes for footnotes[1]. You can change this but the most important thing is to keep the convention consistent throughout the thesis. Footnotes themselves should be full, descriptive sentences (beginning with a capital letter and ending with a full stop). The APA6 states: "Footnote numbers should be superscripted, [...], following any punctuation mark except a dash." The Chicago manual of style states: "A note number should be placed at the end of a sentence or clause. The number follows any punctuation mark except the dash, which it precedes. It follows a closing parenthesis."

The bibliography is typeset with references listed in alphabetical order by the first author's last name. This is similar to the APA referencing style. To see how LaTeX typesets the bibliography, have a look at the very end of this document (or just click on the reference number links in in-text citations).

[1]Such as this footnote, here down at the bottom of the page.

TABLE 1.1: The effects of treatments X and Y on the four groups studied.

Groups	Treatment X	Treatment Y
1	0.2	0.8
2	0.17	0.7
3	0.24	0.75
4	0.68	0.3

A Note on bibtex

The bibtex backend used in the template by default does not correctly handle unicode character encoding (i.e. "international" characters). You may see a warning about this in the compilation log and, if your references contain unicode characters, they may not show up correctly or at all. The solution to this is to use the biber backend instead of the outdated bibtex backend. This is done by finding this in `main.tex`: *backend=bibtex* and changing it to *backend=biber*. You will then need to delete all auxiliary BibTeX files and navigate to the template directory in your terminal (command prompt). Once there, simply type `biber main` and biber will compile your bibliography. You can then compile `main.tex` as normal and your bibliography will be updated. An alternative is to set up your LaTeX editor to compile with biber instead of bibtex, see here for how to do this for various editors.

1.10.4 Tables

Tables are an important way of displaying your results, below is an example table which was generated with this code:

```
\begin{table}
\caption{The effects of treatments X and Y on the four groups studied.}
\label{tab:treatments}
\centering
\begin{tabular}{l l l}
\toprule
\tabhead{Groups} & \tabhead{Treatment X} & \tabhead{Treatment Y} \\
\midrule
1 & 0.2 & 0.8\\
2 & 0.17 & 0.7\\
3 & 0.24 & 0.75\\
4 & 0.68 & 0.3\\
\bottomrule\\
\end{tabular}
\end{table}
```

You can reference tables with `\ref{<label>}` where the label is defined within the table environment. See `Chapter1.tex` for an example of the label and citation (e.g. Table 1.1).

1.10.5 Figures

There will hopefully be many figures in your thesis (that should be placed in the *Figures* folder). The way to insert figures into your thesis is to use a code template like this:

```
\begin{figure}
\centering
\includegraphics{Figures/Electron}
\decoRule
\caption[An Electron]{An electron (artist's impression).}
\label{fig:Electron}
\end{figure}
```

Also look in the source file. Putting this code into the source file produces the picture of the electron that you can see in the figure below.

FIGURE 1.1: An electron (artist's impression).

Sometimes figures don't always appear where you write them in the source. The placement depends on how much space there is on the page for the figure. Sometimes there is not enough room to fit a figure directly where it should go (in relation to the text) and so LaTeX puts it at the top of the next page. Positioning figures is the job of LaTeX and so you should only worry about making them look good!

Figures usually should have captions just in case you need to refer to them (such as in Figure 1.1). The `\caption` command contains two parts, the first part, inside the square brackets is the title that will appear in the *List of Figures*, and so should be short. The second part in the curly brackets should contain the longer and more descriptive caption text.

The `\decoRule` command is optional and simply puts an aesthetic horizontal line below the image. If you do this for one image, do it for all of them.

LaTeX is capable of using images in pdf, jpg and png format.

1.10.6 Typesetting mathematics

If your thesis is going to contain heavy mathematical content, be sure that LaTeX will make it look beautiful, even though it won't be able to solve the equations for you.

The "Not So Short Introduction to LaTeX" (available on CTAN) should tell you everything you need to know for most cases of typesetting mathematics. If you need more information, a much more thorough mathematical guide is available from the AMS called, "A Short Math Guide to LaTeX" and can be downloaded from: ftp://ftp.ams.org/pub/tex/doc/amsmath/short-math-guide.pdf

There are many different LaTeX symbols to remember, luckily you can find the most common symbols in The Comprehensive LaTeX Symbol List.

You can write an equation, which is automatically given an equation number by LaTeX like this:

```
\begin{equation}
E = mc^{2}
\label{eqn:Einstein}
\end{equation}
```

This will produce Einstein's famous energy-matter equivalence equation:

$$E = mc^2 \tag{1.4}$$

All equations you write (which are not in the middle of paragraph text) are automatically given equation numbers by LaTeX. If you don't want a particular equation numbered, use the unnumbered form:

```
\[ a^{2}=4 \]
```

1.11 Sectioning and Subsectioning

You should break your thesis up into nice, bite-sized sections and subsections. LaTeX automatically builds a table of Contents by looking at all the `\chapter{}`, `\section{}` and `\subsection{}` commands you write in the source.

The Table of Contents should only list the sections to three (3) levels. A `chapter{}` is level zero (0). A `\section{}` is level one (1) and so a `\subsection{}` is level two (2). In your thesis it is likely that you will even use a `subsubsection{}`, which is level three (3). The depth to which the Table of Contents is formatted is set within `MastersDoctoralThesis.cls`. If you need this changed, you can do it in `main.tex`.

1.12 In Closing

You have reached the end of this mini-guide. You can now rename or overwrite this pdf file and begin writing your own `Chapter1.tex` and the rest of your thesis. The easy work of setting up the structure and framework has been taken care of for you. It's now your job to fill it out!

Good luck and have lots of fun!

Guide written by —
Sunil Patel: www.sunilpatel.co.uk
Vel: LaTeXTemplates.com

Chapter 2

Chapter Title Here

2.1 Properties of $\backslash c$

To have a clear idea of what we can and need to prove about the algorithms involving Brzozowski's derivatives, there are a few properties we need to be clear about it.

2.1.1 function $\backslash c$ is not 1-to-1

The derivative *w.r.t* character c is not one-to-one. Formally,
$$\exists r_1\ r_2 . r_1 \neq r_2 and r_1 \backslash c = r_2 \backslash c$$

This property is trivially true for the character regex example:

$$r_1 = e;\ r_2 = d;\ r_1 \backslash c = \mathbf{0} = r_2 \backslash c$$

But apart from the cases where the derivative output is **0**, are there non-trivial results of derivatives which contain strings? The answer is yes. For example,

Let $r_1 = a^* b \quad r_2 = (a \cdot a^*) \cdot b + b.$
where a is not nullable.
$$r_1 \backslash c = ((a \backslash c) \cdot a^*) \cdot c + b \backslash c$$
$$r_2 \backslash c = ((a \backslash c) \cdot a^*) \cdot c + b \backslash c$$

We start with two syntactically different regexes, and end up with the same derivative result, which is a "meaningful" regex because it contains strings. We have rediscovered Arden's lemma:

$$A^* B = A \cdot A^* \cdot B + B$$

2.1.2 Subsection 1

Nunc posuere quam at lectus tristique eu ultrices augue venenatis. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aliquam erat volutpat. Vivamus sodales tortor eget quam adipiscing in vulputate ante ullamcorper. Sed eros ante, lacinia et sollicitudin et, aliquam sit amet augue. In hac habitasse platea dictumst.

2.1.3 Subsection 2

Morbi rutrum odio eget arcu adipiscing sodales. Aenean et purus a est pulvinar pellentesque. Cras in elit neque, quis varius elit. Phasellus fringilla, nibh eu tempus venenatis, dolor elit posuere quam, quis adipiscing urna leo nec orci. Sed nec nulla auctor odio aliquet consequat. Ut nec nulla in ante ullamcorper aliquam at sed dolor. Phasellus fermentum magna in augue gravida cursus. Cras sed pretium lorem. Pellentesque eget ornare odio. Proin accumsan, massa viverra cursus pharetra, ipsum nisi lobortis velit, a malesuada dolor lorem eu neque.

2.2 Main Section 2

Sed ullamcorper quam eu nisl interdum at interdum enim egestas. Aliquam placerat justo sed lectus lobortis ut porta nisl porttitor. Vestibulum mi dolor, lacinia molestie gravida at, tempus vitae ligula. Donec eget quam sapien, in viverra eros. Donec pellentesque justo a massa fringilla non vestibulum metus vestibulum. Vestibulum in orci quis felis tempor lacinia. Vivamus ornare ultrices facilisis. Ut hendrerit volutpat vulputate. Morbi condimentum venenatis augue, id porta ipsum vulputate in. Curabitur luctus tempus justo. Vestibulum risus lectus, adipiscing nec condimentum quis, condimentum nec nisl. Aliquam dictum sagittis velit sed iaculis. Morbi tristique augue sit amet nulla pulvinar id facilisis ligula mollis. Nam elit libero, tincidunt ut aliquam at, molestie in quam. Aenean rhoncus vehicula hendrerit.

Chapter 3

Common Identities In Simplification-Related Functions

3.1 Idempotency of *simp*

$$\mathit{simp}\ r = \mathit{simp}\ \mathit{simp}\ r \tag{3.1}$$

This property means we do not have to repeatedly apply simplification in each step, which justifies our definition of *blexer_simp*. It will also be useful in future proofs where properties such as closed forms are needed. The proof is by structural induction on *r*.

3.1.1 Syntactic Equivalence Under *simp*

We prove that minor differences can be annhilated by *simp*. For example,

$$\begin{gathered}\mathit{simp}\ (\mathit{simp\_ALTs}\ (\mathit{map}\ (\_\backslash\ x)\ (\mathit{distinct}\ \mathit{rs}\ \phi))) = \\ \mathit{simp}\ (\mathit{simp\_ALTs}\ (\mathit{distinct}\ (\mathit{map}\ (\_\backslash\ x)\ \mathit{rs})\ \phi))\end{gathered}$$

3.1.2 Subsection 2

Morbi rutrum odio eget arcu adipiscing sodales. Aenean et purus a est pulvinar pellentesque. Cras in elit neque, quis varius elit. Phasellus fringilla, nibh eu tempus venenatis, dolor elit posuere quam, quis adipiscing urna leo nec orci. Sed nec nulla auctor odio aliquet consequat. Ut nec nulla in ante ullamcorper aliquam at sed dolor. Phasellus fermentum magna in augue gravida cursus. Cras sed pretium lorem. Pellentesque eget ornare odio. Proin accumsan, massa viverra cursus pharetra, ipsum nisi lobortis velit, a malesuada dolor lorem eu neque.

3.2 Main Section 2

Sed ullamcorper quam eu nisl interdum at interdum enim egestas. Aliquam placerat justo sed lectus lobortis ut porta nisl porttitor. Vestibulum mi dolor, lacinia molestie gravida at, tempus vitae ligula. Donec eget quam sapien, in viverra eros. Donec pellentesque justo a massa fringilla non vestibulum metus vestibulum. Vestibulum in orci quis felis tempor lacinia. Vivamus ornare ultrices facilisis. Ut hendrerit volutpat vulputate. Morbi condimentum venenatis augue, id porta ipsum vulputate in. Curabitur luctus tempus justo. Vestibulum risus lectus, adipiscing nec condimentum quis, condimentum nec nisl. Aliquam dictum sagittis velit sed iaculis. Morbi tristique augue sit amet nulla pulvinar id facilisis ligula mollis. Nam elit libero, tincidunt ut aliquam at, molestie in quam. Aenean rhoncus vehicula hendrerit.

Appendix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:
`\hypersetup{urlcolor=red}`, or
`\hypersetup{citecolor=green}`, or
`\hypersetup{allcolor=blue}`.
If you want to completely hide the links, you can use:
`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.
If you want to have obvious links in the PDF but not the printed text, use:
`\hypersetup{colorlinks=false}`.

Bibliography

Brzozowski, J. A. (1964). “Derivatives of Regular Expressions”. In: *Journal of the ACM* 11.4, pp. 481–494.

Coquand, T. and V. Siles (2011). “A Decision Procedure for Regular Expression Equivalence in Type Theory”. In: *Proc. of the 1st International Conference on Certified Programs and Proofs (CPP)*. Vol. 7086. LNCS, pp. 119–134.

Krauss, A. and T. Nipkow (2012). “Proof Pearl: Regular Expression Equivalence and Relation Algebra”. In: *Journal of Automated Reasoning* 49, pp. 95–106.

Owens, S. and K. Slind (2008). “Adapting Functional Programs to Higher Order Logic”. In: *Higher-Order and Symbolic Computation* 21.4, pp. 377–409.

Ribeiro, Rodrigo and André Du Bois (2017). “Certified Bit-Coded Regular Expression Parsing”. In: *Proceedings of the 21st Brazilian Symposium on Programming Languages*. SBLP 2017. Fortaleza, CE, Brazil: Association for Computing Machinery. ISBN: 9781450353892. DOI: 10.1145/3125374.3125381. URL: https://doi.org/10.1145/3125374.3125381.