POSIX Regular Expression Matching and Lexing

Tan, Chengsong

Awarding institution:
King's College London

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 Group
Department or Informatics

September 16, 2024

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 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:

Abstract

POSIX is the most widely used disambiguation strategy for regular expression matching. There are some difficulties associated with the POSIX strategy and according to tests conducted by Kulkewitz, many regular expression matchers implementing this strategy produce incorrect results. This thesis is concerned with an POSIX regular expression matching algorithm introduced by Sulzmann and Lu. This algorithm uses bitcoded regular expressions and is based on the idea of Brzozowski derivatives. The algorithm generates POSIX values which encode the information of how a regular expression matches a string - that is, which part of the string is matched by which part of the regular expression. This information is needed in the context of lexing in order to extract and to classify tokens.

While a formalised correctness proof for Sulzmann and Lu's algorithm already exists, this proof does not include any of the crucial simplification rules. These simplification rules are however necessary in order to have an acceptable runtime for this algorithm. Our version of the simplification rules includes a number of fixes and improvements: one problem we fix has to do with their use of the nub function that does not remove non-trivial duplicates. We improve the simplification rules by formulating them as simple recursive function and also by simplifying more instances of regular expressions. As a result we can establish a bound on the size of derivatives. Our proofs are formalised in Isabelle/HOL.

Acknowledgements

I would like to express my deepest thanks to my supervisor Doctor Christian Urban, who have been always extremely supportive thoughout my PhD, in all sorts of ways. Supervisionwise, Christian always thinks in terms of the best interests for the student, to which I am eternally grateful for. I would also like to thank Doctor Ning Zhang, who have always been very gentle and caring to me, quick to lend a helping hand at difficult times. I want to thank Doctor Kathrin Stark, my SIGPLAN mentor, for offering brilliant advice at the late stage of my PhD. My transition from a PhD student to a postdoc researcher could not have been so smooth without Kathrin's mentoring.

I gratefully thank the financial support from the King's-CSC Scholarship. I want to thank Jeanna Wheeler for helping me with keeping sane during my time during the PhD and COVID times when an encouraging and compassionate person was very appreciated.

I want to thank my father Haiyan Tan and my mother Yunan Cheng, for their unconditional love, and who I have not seen face to face for three years. I really miss you. I want to thank my friends Yuying Chen, Kai Zeng, Rui Luo, Jingyi Liu, Qingtian Ye, Yaoyi Li, and many others, who have always been very patient and compassionate, giving clever advice when I turned to them for help.

Contents

To my granny, Shoucai Cheng.
You are dearly missed.

Chapter 1

Introduction

Regular expressions, since their inception in the 1950s [48], have been subject to extensive study and implementation. Their primary application lies in text processing–finding matches and identifying patterns in a string. For example, a simple regular expression that tries to recognise email addresses is

```
[a-z0-9._]+@[a-z0-9.-]+\.[a-z]{2,6}
```

This expression assumes all letters in the email have been converted into lower-case. The local-part is recognised by the first bracketed expression $[a-z0-9.\_]^+$ where the operator "+" (should be viewed as a superscript) means that it must be some non-empty string made of alpha-numeric characters. After the "@" sign is the sub-expression that recognises the domain of that email, where $[a-z]^{\{2,6\}}$ specifically matches the top-level domain, such as "org", "com", "uk" and etc. The counters in the superscript such as 2 and 6 specify that all top-level domains should have between two to six characters. This regular expression does not represent all possible email addresses (e.g. those involving "-" cannot be recognised), but patterns of similar shape and using roughly the same set of syntax are used everywhere in our daily life, for example in compilers [55], networking [72], software engineering (IDEs) [12] and operating systems [40], where the correctness of matching can be crucially important.

Implementations of regular expression matching out in the wild, however, are surprisingly unreliable. An example is the regular expresion $(a^*)^*b$ and the string $aa\ldots a$. The expectation is that any regex engine should be able to solve this (return a no match) in no time. However, if this is tried out in an regex engine like that of Java 8, the runtime would quickly grow beyond 100 seconds with just dozens of characters Such behaviour can result in Denial-of-Service (ReDoS) attacks with minimal resources–just the pair of "evil" regular expression and string. The outage of the web service provider Cloudflare [1] in 2019 [2] is a recent example where such issues caused serious negative impact.

The reason why these regex matching engines get so slow is because they use backtracking or a depth-first-search (DFS) on the search space of possible matches. They employ backtracking algorithms to support back-references, a mechanism allowing expressing languages which repeating an arbitrary long string such as $\{ww|w \in \Sigma*\}$. Such a constructs makes matching NP-complete, for which no known non-backtracking algorithms exist. More modern programming languages' regex engines such as Rust and GO's do promise linear-time behaviours with respect to input string, but they do not support back-references, and often impose ad-hoc restrictions on the input patterns. More importantly, these promises are purely empirical, making them prone to future ReDoS attacks and other types of errors.

The other unreliability of industry regex engines is that they do not produce the desired output. Kuklewicz have found out during his testing of practical regex engines that almost all of them are incorrect with respect to the POSIX standard, a disambiguation strategy adopted most widely in computer science. Roughly speaking POSIX lexing means to always go for the longest initial submatch possible, for instance a single iteration aa is preferred over two iterations a, a when matching $(a|aa)^*$ with the string aa. The POSIX strategy as summarised by Kuklewicz goes as follows:

- regular expressions (REs) take the leftmost starting match, and the longest match starting there earlier subpatterns have leftmost-longest priority over later subpatterns
- higher-level subpatterns have leftmost-longest priority over their component subpatterns
- ...

The author noted that various lexers that come with POSIX operating systems are rarely correct according to this standard. A test case that exposes the bugs is the regular expression $(aba|ab|a)^*$ and string $ababa$. A correct match would split the string into ab, aba, involving two repetitions of the Kleene star $(aba|ab|a)^*$. Most regex engines would instead return two (partial) matches aba and a [1]. There are numerous occasions where programmers realised the subtlety and difficulty to implement POSIX correctly, one such quote from Go's regexp library author: [2]

" The POSIX rule is computationally prohibitive and not even well-defined. "

These all point towards a formal treatment of POSIX lexing to clear up inaccuracies and errors in understanding and implementation of regex. The work presented in this thesis uses formal proofs to ensure the correctness and runtime properties of POSIX regular expression lexing.

Formal proofs or mechanised proofs are computer checked programs that certify the correctness of facts with respect to a set of axioms and definitions. They provide an unprecendented level of asssurance that an algorithm will perform as expected under all inputs. We believe such proofs can help eliminate catastrophic backtracking once and for all. The software systems that help people interactively build and check formal proofs are called proof assitants or interactive theorem provers. Isabelle/HOL [64] is a widely-used proof assistant with a simple type theory and powerful automated proof generators like sledgehammer. We chose to use Isabelle/HOL for its powerful automation and ease and simplicity in expressing regular expressions and regular languages.

The algorithm we work on is based on Brzozowski derivatives. Brzozowski invented the notion of "derivatives" on regular expressions [23], and the idea is that we can reason about what regular expressions can match by taking derivatives of them. A derivative operation takes a regular expression r and a character c, and returns a new regular expression $r\backslash c$. The derivative is taken with respect to c: it transforms r in such a way that the resulting derivative $r\backslash c$ contains all strings in the language of r that starts with c, but now with the head character c thrown away. For

[1] Try it out here: https://regex101.com/r/c5hga5/1, last accessed 22-Aug-2023
[2] https://pkg.go.dev/regexp#pkg-overview, last accessed 22-Aug-2023

example, for r equal to $(aba|ab|a)^*$ as discussed earlier, its derivative with respect to character a is

$$r\backslash a = (ba|b|\mathbf{1})(aba|ab|a)^*.$$

Brzozowski derivatives are purely algebraic operations that can be implemented as recursive functions which do pattern matches on the structure of the regular expression. For example, the derivatives of character regular expressions, Kleene star and bounded repetitions can be described by the following code equations:

$$\begin{array}{lcl} d\backslash c & \stackrel{\text{def}}{=} & \textit{if } c = d \textit{ then } \mathbf{1} \textit{ else } \mathbf{0} \\ (r^*)\backslash c & \stackrel{\text{def}}{=} & (r\backslash c) \cdot r^* \\ r^{\{n\}}\backslash c & \stackrel{\text{def}}{=} & r\backslash c \cdot r^{\{n-1\}} (\text{when } n > 0) \end{array}$$

Taking derivatives repeatedly with respect to the sequence of characters in the string s, one obtain a regular expression $r\backslash s$ containing the information of how r matched s, and this can be defined recursively as well: $r\backslash(cs) \stackrel{\text{def}}{=} (r\backslash c)\backslash s$. One can prove with straightforward induction on s that a matcher based on derivatives correctly calculates whether $s \in L\ r$ by judging whether the empty string is in $L\ (r\backslash s)$. In other words,

$$s \in L\, r \quad \text{if and only if} \quad [] \in r\backslash s.$$

Thanks to such theorem prover friendly definitions, there have been a sizable number of formalisations of derivatives and regular expressions matching in different theorem provers (e.g. [66] [62] and [27]) in the formal reasoning community.

The variant of the problem we are looking at is using derivatives to calculate lexical values. We build on the work of Sulzmann and Lu [77]. The idea is that from the final derivative $r\backslash s$ more information can be computed than just a YES/NO answer. One can extract a value for *how* $r\backslash s$ matched the empty string and then incrementally build on that value by a reverse operation of derivatives called *injection*. After all characters have been injected back, the lexing tree is also built for how r matched s. For instance, $a^*\backslash a = \mathbf{1} \cdot a^*$ can be turned into a value $Seq\ Empty\ (Stars\, Nil)$, and this value can be injected into the character a to form $Stars\,[a]$. The value $Stars\ [a]$ tells us how a^* matched the single-character string a. This procedure can be further simplified by eliminating the injecting back characters phase, and constructing lexing information incrementally while derivatives are taken. The lexing information is attached to regular expressions as *bitcodes*. On the same example again it would be $a^*\backslash a =_1 \mathbf{1} \cdot a^*$. Using the attached bitcode $_1$ one knows it should be a single iteration of a^*, and therefore the value $Stars\ [a]$. Ausaf et al. [13] [14] have formalised the definition of a POSIX value and proved that the algorithms we just described produced POSIX values. However, it is not clear that an even further optimised version of lexing algorithm with bitcodes described in [77] is correct. The authors believed that their optimised bitcoded lexer is able to compute a lexical tree in linear time with respect to the input string length. As they put it,

> "Hence, we can argue that the above mentioned functions/operations have constant time complexity which implies that we can incrementally compute bitcoded parse trees in linear time in the size of the input."

We show in this thesis with examples that their assumption "operations have constant time complexity" is untrue. Each derivative operation might take more time than its previous step as the internal data structure grows unbounded, and the algorithm has an exponential worst-case complexity. We then present an improved optimised bitcoded algorithm while we were seeking to remove their complexity bug.

We call our improved procedure *blexer_simp*. We prove the correctness of *blexer_simp* by showing it produces the same output as the un-optimised lexer *blexer* that Ausaf et al. [14] proved correct. This allows us to utilise the correctness results they already have. The proof of *blexer_simp* itself is entirely different from *blexer*, as the simplifications destroy the structure of the regular expressions, upon which the structural induction used in the correctness proof of *blexer* depends.

The central idea of the correctness of *blexer_simp* is that the intermediate regular expressions calculated by *blexer_simp* contains strictly less information than those of *blexer*'s, but still enough to produce a POSIX value. To capture this relation with *blexer* and *blexer_simp*, we defined a term-rewriting relation between unsimplified and simplified regular expressions. We prove that one can always rewrite in finitely many steps from an intermediate regular expression in *blexer* to the intermediate regular expression in *blexer_simp*, if the same string input has been provided. We then prove that for any two regular expressions that can be rewritten to one anther, it is always possible to extract the same bitcodes containing the POSIX values.

We then use the idea of our rewriting relation in a different setting: we show that our *blexer_simp* algorithm fulfills Sulzmann and Lu's claim for their original optimised algorithm, that the size of the regular expressions in each derivative operation stays below a constant. We prove this by showing that derivatives with simplifications are equal to a "closed form" of regular expressions. The rewriting relation serves as bridges between the intermediate regular expressions of *blexer_simp* and their closed forms. This space complexity result implies that the improved lexer we have actually enjoys linear complexity, though filling the last gap is future work.

To summarise, we expect modern regex matching and lexing engines to be reliabe, fast and correct, and support rich syntax constructs like bounded repetitions and back references. Backtracking regex engines have exhibited exponential worst-case behaviours (catastrophic backtracking) for employing a depth-first-search on the search tree of possible matches. To ensure the correctness and predictable behaviour of lexical analysis, we propose to build a formally verified lexer that satisfy correctness and non-backtracking requirements in a bottom-up manner using Brzozowski derivatives. Derivatives on regular expressions are popular in the theorem proving community because their algebraic features are amenable to formal reasoning. a derivative-based matching algorithm avoids backtracking, by explicitly representing possible matches on the same level of a search tree as regular expression terms in a breadth-first manner. Efficiency of such algorithms rely on limiting the size of the derivatives during the computation, for example by eliminating redundant terms that lead to identical matches. We build on the line of work by Ausaf et al. and Sulzmann and Lu. The end result is a formally verified lexer that comes with additional formal guarantees on space complexity of derivatives.

1.1 Contribution and Related Work

We have made mainly two contributions in this thesis: proving the lexer *blexer_simp* is both i) correctness and ii)fast. It is correct w.r.t a formalisation of POSIX lexing by Ausaf et al.[14]. It is fast compared with the optimised implementations of Sulzmann and Lu's orignal paper. In their work, Sulzmann and Lu thought that their algorithm is linear and each operation takes constant time. We show that this is not true by giving a counterexample, in which the data structure size is ever-increasing. We then formally prove that we fix their complexity bug in our algorithm, which indeed

achieves their constant-time operation claim. We present a brief survey of the related work here, providing necessary context for where our contribution lies.

1.1.1 Regular Expressions, Derivatives and POSIX Lexing

Regular expressions were introduced by Kleene in the 1950s [49]. Since then they have become a fundamental concept in formal languages and automata theory [73]. Brzozowski defined derivatives on regular expressions in his PhD thesis in 1964 [23], in which he proved the finiteness of the numbers of regular expression derivatives modulo the ACI-axioms. It is worth pointing out that this result does not directly translate to our finiteness proof, and our proof does not depend on it. The key observation is that our version of the Sulzmann and Lu's algorithm [77] represents derivative terms in a way that allows efficient de-duplication, and we do not make use of an equivalence checker that exploits the ACI-equivalent terms.

Central to this thesis is the work by Sulzmann and Lu [77]. They first introduced the elegant and simple idea of injection-based lexing. The second algorithm in this paper is about bit-coded lexing, which used an idea from Nielsen and Henglein [54]. In a follow-up work [78], Sulzmann and Steenhoven incorporated these ideas into a tool called *dreml*. The pencil-and-paper proofs in [77] based on the ideas by Frisch and Cardelli [39] were later found to have unfillable gaps by Ausaf et al. [14], who came up with an alternative proof inspired by Vansummeren [84]. Sulzmann and Thiemann extended the Brzozowski derivatives to shuffling regular expressions [79], which are a very succinct way of describing regular expressions.

Regular expressions and lexers have been a popular topic among the theorem proving and functional programming community. In the next section we give a list of lexers and matchers that come with a machine-checked correctness proof.

1.1.2 Matchers and Lexers with Mechanised Proofs

We are aware of a mechanised correctness proof of Brzozowski's derivative-based matcher in HOL4 by Owens and Slind [66]. Another one in Isabelle/HOL is part of the work by Krauss and Nipkow [50]. Another one in Coq is given by Coquand and Siles [27]. Also Ribeiro and Du Bois gave one in Agda [71]. The most recent works on verified lexers to our best knowledge are the Verbatim [32] and Verbatim++ [33] lexers by Egolf et al. Verbatim is based on derivatives but does not simplify them. Therefore it can be very slow on evil regular expressions. The Verbatim++ lexer adds many correctness-preserving optimisations to the Verbatim lexer, and is therefore quite fast on many inputs. The problem is that Egolf et al. chose to use traditional DFAs to speed up lexing, but then dealing with bounded repetitions is a real bottleneck.

This thesis builds on the formal specifications of POSIX rules and formal proofs by Ausaf et al. [14]. The bounded repetitions presented here is a continuation of the work by Ausaf [13]. Proving *blexer_simp* requires a different proof strategy compared to that by Ausaf [13]. We invent a rewriting relation as an inductive predicate which captures a strong enough invariance that ensures correctness, which commutes with the derivative operation. This predicate allows a simple induction on the input string to go through. This method is general and extensible, and we show how it helps already with the proof on the space complexity.

Automata formalisations are in general harder and more cumbersome to deal with for theorem provers [63]. To represent them, one way is to use graphs, but graphs are not inductive datatypes. Having to set the inductive principle on the

number of nodes in a graph makes formal reasoning non-intuitive and convoluted, resulting in large formalisations [80]. When combining two graphs, one also needs to make sure that the nodes in both graphs are distinct. If they are not distinct, then renaming of the nodes is needed. There are some more clever representations, for example one by Paulson using hereditarily finite sets [67]. There the problem with combining graphs can be solved better. The *FinType* datatype from *ssreflect* can potentially make things slightly easier, as testified by [47].

Another representation for automata are matrices. But the induction for them is not as straightforward either. Both approaches have been used in the past and resulted in huge formalisations.

Because of these problems with automata, we prefer regular expressions and derivatives which can be implemented in theorem provers and functional programming languages with ease.

1.1.3 Different Definitions of POSIX Rules

There are different ways to formalise values and POSIX matching. In this thesis we choose to work on the formal definitions given by Ausaf et al.[14].

Cardelli and Frisch [39] have developed a notion of *non-problematic values* which is a slight variation of the values in this thesis.They then defined an ordering between values, and showed that the maximal element of those values correspond to the output of their GREEDY lexing algorithm.

Okui and Suzuki [65] allow iterations of values to flatten to an empty string. They refer to the more restrictive version as used in this thesis (which was defined by Ausaf et al. [14]) as *canonical values*. The very interesting link between the work by Ausaf et al. and Okui and Suzuki is that they have distinct formalisations of POSIX values, and yet they define the same notion. See [14] for details of the alternative definitions given by Okui and Suzuki and the formalisation described in this thesis.

Sulzmann and Lu themselves have come up with a POSIX definition [77]. In their paper they defined an ordering between values with respect to regular expressions, and tried to establish that their algorithm outputs the minimal element by a pencil-and-paper proof. But having the ordering relation taking regular expression as parameters causes the transitivity of their ordering to not go through.

1.1.4 Static Analysis of Evil Regex Patterns

When faced with catastrophic backtracking, sometimes programmers have to rewrite their regexes in an ad hoc fashion. Knowing which patterns should be avoided can be helpful. Animated tools to "debug" regular expressions such as [17] [35] are also popular. Weideman [85] came up with non-linear polynomial worst-time estimates for regexes and “attack string” that exploit the worst-time scenario, and introduced "attack automaton" that generates attack strings.

Static analysis tools can detect “evil” patterns before they were executed. For instance, Minamide et al. [58] have used tree transducers to analyse the complexity of “evil” regular expressions, and such analysis is sound and complete. The drawback is that such procedures can be exponential in the worst case. We are also aware of a similar line of research on this [69]. Sometimes these static analyzers over-approximate evil regular expression patterns, and there can be many false positives [29]. In general the static analysis of regular expressions is hard and worst-case exponential.

1.1.5 Optimisations

Perhaps the biggest problem that prevents derivative-based lexing from being more widely adopted is that they tend to be not very fast in practice, unable to reach throughputs like gigabytes per second, which is the application we have in mind when we started looking at the topic. Commercial regular expression matchers such as Snort [72] and Bro [68] are capable of inspecting payloads at line rates (which means up to a few gigabits per second) against thousands of regex rules [76]. For our algorithm to be more attractive for practical use, we need more correctness-preserving optimisations.

FPGA-based implementations such as [75] have the advantages of being reconfigurable and parallel, but suffer from lower clock frequency and scalability. Traditional automaton approaches that use DFAs instead of NFAs benefit from the fact that only a single transition is needed for each input character [18]. Lower memory bandwidth leads to faster performance. However, they suffer from exponential memory requirements on bounded repetitions. Compression techniques are used, such as those in [53] and [19]. Variations of pure NFAs or DFAs like counting-set automata [82] have been proposed to better deal with bounded repetitions. But they usually do not contain any formalised proofs.

Another direction of optimisation for derivative-based approaches is defining string derivatives directly, without recursively decomposing them into character-by-character derivatives. For example, instead of replacing $(r_1 + r_2)\backslash(c :: cs)$ by $((r_1 + r_2)\backslash c)\backslash cs$, we rather calculate $(r_1\backslash(c :: cs) + r_2\backslash(c :: cs))$. This has the potential to speed up matching because input is processed at a larger granularity. One interesting point is to explore whether this can be done to *inj* as well, so that we can generate a lexical value rather than simply get a matcher. It is also not yet clear how this idea can be extended to sequences and stars.

1.1.6 Derivatives and Zippers

Zippers are a data structure designed to focus on and navigate between local parts of a tree. The idea is that often operations on large trees only deal with local regions each time. Therefore it would be a waste to traverse the entire tree if the operation only involves a small fraction of it. Zippers were first formally described by Huet [44]. Typical applications of zippers involve text editor buffers and proof system databases. In our setting, the idea is to compactify the representation of derivatives with zippers, thereby making our algorithm faster. Below we describe several works on parsing, derivatives and zippers.

Edelmann et al. developed a formalised parser for LL(1) grammars using derivatives [31]. They adopted zippers to improve the speed, and argued that the runtime complexity of their algorithm was linear with respect to the input string. They did not provide a formal proof for this.

The idea of using Brzozowski derivatives on general context-free parsing was first implemented by Might et al. [57]. They used memoization and fixpoint constructions to eliminate infinite recursion, which is a major problem for using derivatives with context-free grammars. Their initial version was quite slow—-exponential on the size of the input. Adams et al. [6] improved the speed and argued that their version was cubic with respect to the input. Darragh and Adams [28] further improved the performance by using zippers in an innovative way–their zippers had multiple focuses instead of just one in a traditional zipper to handle ambiguity. Their algorithm was not formalised though.

1.1.7 Back-References

Bakc-references syntax in a regex engine allows html-style paired tags to be represented. We adopt the common practice of calling them rewbrs (Regular Expressions With Back References) for brevity. It has been shown by Aho [7] that the k-vertex cover problem can be reduced to the problem of rewbrs matching, and therefore matching with rewbrs is NP-complete. Another reduction from 3-SAT problem to rewbrs can be found in [45]. Given the depth of the problem, the progress made at the full generality of arbitrary rewbrs matching has been slow with theoretical work on them being fairly recent.

Campaneu et al. studied rewbrs in the context of formal languages theory in [24]. They devised a pumping lemma for Perl-like regexes, proving that the languages denoted by them is properly contained in context-sensitive languages. More interesting questions such as whether the languages denoted by Perl-like regexes can express palindromes ($\{w \mid w = w^R\}$) are discussed in [25] and [26]. Freydenberger [37] investigated the expressive power of back-references. He showed several undecidability and decriptional complexity results for back-references, concluding that they add great power to certain programming tasks but are difficult to comprehend. An interesting question would then be whether we can add restrictions to them, so that they become expressive enough for practical use, such as matching html files, but not too powerful. Freydenberger and Schmid [38] introduced the notion of deterministic regular expressions with back-references to achieve a better balance between expressiveness and tractability.

Fernau and Schmid [34] and Schmid [74] investigated the time complexity of different variants of back-references. We are not aware of any work that uses derivatives with back-references.

The recently published work on formally verified lexer [60] by Moseley et al. moves a big step forward in practical performance–their lexer can compete with unverified commercial regex engines like those in Java, Python and etc. and supports lookaheads. To reach their speed on larger datasets like [3], one has to incorporate many fine-tunings and one of the first modifications needed would be to drop whole-string lexing but only extract the submatch needed by the user. The only drawback of the work seems that the underlying formal proofs is still under development.

1.1.8 Complexity Results

Our formalisation of space complexity is unique among similar works in the sense that to our knowledge there are not other certified lexing/parsing algorithms with similar data structure size bound theorems. Common practices involve making empirical analysis of the complexity of the algorithm in question ([32], [33]), or relying on prior (unformalised) complexity analysis of well-known algorithms ([83]), making them prone to complexity bugs.

Whilst formalised complexity theorems have not yet appeared in other certified lexers/parsers, they do find themselves in the broader theorem-proving literature: *time credits* have been formalised for separation logic in Coq [11] to characterise the runtime complexity of union-find. The idea is that the total number of instructions executed is equal to the time credits consumed during the execution of an algorithm. Armaël et al. [42] have extended the framework to allow expressing time credits using big-O notations, so one can prove both the functional correctness and asymptotic

complexity of higher-order imperative algorithms. A number of formal proofs also exist for some other algorithms in Coq [21].

The big-O notation have also been formalised in Isabelle [16] . Our work focuses on the space complexity of the algorithm under our notion of the size of a regular expression. Despite not being a direct asymptotic time complexity proof, our result is an important stepping stone towards one.

Brzozowski showed that there are finitely many similar deriviatives, where similarity is defined in terms of ACI equations. This allows him to use derivatives as a basis for DFAs where each state is labelled with a derivative. However, Brzozowski did not show anything about the size of the derivatives. Antimirov showed that there can only be finitely many partial derivatives for a regular expression and any set of strings. He showed that the number is actually the "alphabetical width" plus 1. From this result one can relatively easily establish that the size of the partial derivatives is no bigger than $(size\ r)^3$ for every string. Unfortunately this result does not seem to carry over to our setting because partial derivatives have the simplification

$$(r_1 + r_2) \cdot r_3 \rightarrow (r_1 \cdot r_3) + (r_2 \cdot r_3) \tag{1.1}$$

built in. We cannot have this because otherwise we would lose the POSIX property. For instance, the lexing result of regular expression

$$(a + ab) \cdot (bc + c)$$

with respect to string *abc* using our lexer with the simplification rule 1.1 would be

$$Left(Seq\ (Char\ a), Seq(Char\ b)\ (Char\ c))$$

instead of the correct POSIX value

$$Seq\ (Right\ (Seq\ (Char\ a)\ (Char\ b)))\ (Char\ )$$

Our result about the finite bound also does not say anything about the number of derivatives. In fact there are infinitely many derivatives in general because in the annotated regular expression for STAR we record the number of iterations. What our result shows that the size of the derivatives is bounded, not the number.

In particular, the main problem we solved on top of previous work was coming up with a formally verified algorithm called *blexer_simp* based on Brzozowski derivatives. It calculates a POSIX lexical value from a string and a regular expression. This algorithm was originally by Sulzmann and Lu [77], but we made the key observation that its *nub* function does not really simplify intermediate results where it needs to and improved the algorithm accordingly. We have proven our *blexer_simp*'s internal data structure does not grow beyond a constant N_r depending on the input regular expression r, thanks to the aggressive simplifications of *blexer_simp*:

$$|blexer\_simp\ r\ s| \leq N_r$$

The simplifications applied in each step of *blexer_simp*

$$blexer\_simp$$

keeps the derivatives small, but presents a challenge

establishing a correctness theorem of the below form:

If the POSIX lexical value of matching regular expression r with string s is v, then *blexer_simp r s* = *Some v*. Otherwise *blexer_simp r s* = *None*.

The result is

- an improved version of Sulzmann and Lu's bit-coded algorithm using derivatives with simplifications, accompanied by a proven correctness theorem according to POSIX specification given by Ausaf et al. [14],
- a complexity-related property for that algorithm saying that the internal data structure will remain below a finite bound,
- and an extension to the bounded repetition constructs with the correctness and finiteness property maintained.

With a formal finiteness bound in place, we can greatly reduce the attack surface of servers in terms of ReDoS attacks. The Isabelle/HOL code for our formalisation can be found at

`https://github.com/hellotommmy/posix`

Further improvements to the algorithm with an even stronger version of simplification can be made. We conjecture that the resulting size of derivatives can be bounded by a cubic bound w.r.t. the size of the regular expression. We are working to improve the formalisation, and therefore this is not yet on the AFP. We will give relevant code in Scala, but do not give a formal proof for that in Isabelle/HOL. This is still future work.

1.2 Structure of the thesis

Before talking about the formal proof of *blexer_simp*'s correctness, which is the main contribution of this thesis, we need to introduce two formal proofs which belong to Ausafe et al. In chapter 3 we will introduce the concepts and notations we use for describing regular expressions and derivatives, and the first version of Sulzmann and Lu's lexing algorithm without bitcodes (including its correctness proof). We will give their second lexing algorithm with bitcodes in 4 together with the correctness proof by Ausaf and Urban. Then we illustrate in chapter 5 how Sulzmann and Lu's simplifications fail to simplify correctly. We therefore introduce our own version of the algorithm with correct simplifications and their correctness proof. In chapter 6 we give the second guarantee of our bitcoded algorithm, that is a finite bound on the size of any regular expression's derivatives. We also show how one can extend the algorithm to include bounded repetitions. In chapter 7 we discuss stronger simplification rules which improve the finite bound to a cubic bound. Chapter 8 concludes and mentions avenues of future research.

Chapter 2

Technical Overview

We start with a technical overview of the catastrophic backtracking problem, motivating rigorous approaches to regular expression matching and lexing. In doing so we also briefly introduce common terminology such as bounded repetitions and back-references.

2.1 Motivating Examples

Consider for example the regular expression $(a^*)^*\,b$ and strings of the form $aa..a$. These strings cannot be matched by this regular expression: obviously the expected b in the last position is missing. One would assume that modern regular expression matching engines can find this out very quickly. Surprisingly, if one tries this example in JavaScript, Python or Java 8, even with small strings, say of lenght of around 30 a's, the decision takes large amounts of time to finish. This is inproportional to the simplicity of the input (see graphs in figure 2.1). The algorithms clearly show exponential behaviour, and as can be seen is triggered by some relatively simple regular expressions. Java 9 and newer versions mitigates this behaviour by several magnitudes, but are still slow compared with the approach we are going to use in this thesis.

This superlinear blowup in regular expression engines has caused grief in "real life" where it is given the name "catastrophic backtracking" or "evil" regular expressions. A less serious example is a bug in the Atom editor: a user found out when writing the following code:

```
vVar.Type().Name() == "" && vVar.Kind() == reflect.Ptr
&& vVar.Type().Elem().Name() == "" && vVar.Type().Elem().Kind() ==
reflect.Slice
```

it took the editor half a hour to finish computing. This was subsequently fixed by Galbraith [4], and the issue was due to the regex engine of Atom. But evil regular expressions can be more than a nuisance in a text editor: 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 the web servers to grind to a halt. In this example, the time needed to process the string was $O(n^2)$ with respect to the string length n. This quadratic overhead was enough for the homepage of Stack Exchange to respond so slowly that the load balancer assumed a *DoS* attack and therefore stopped the servers from responding to any requests. This made the whole site become unavailable.

[1] https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016(Last accessed in 2019)

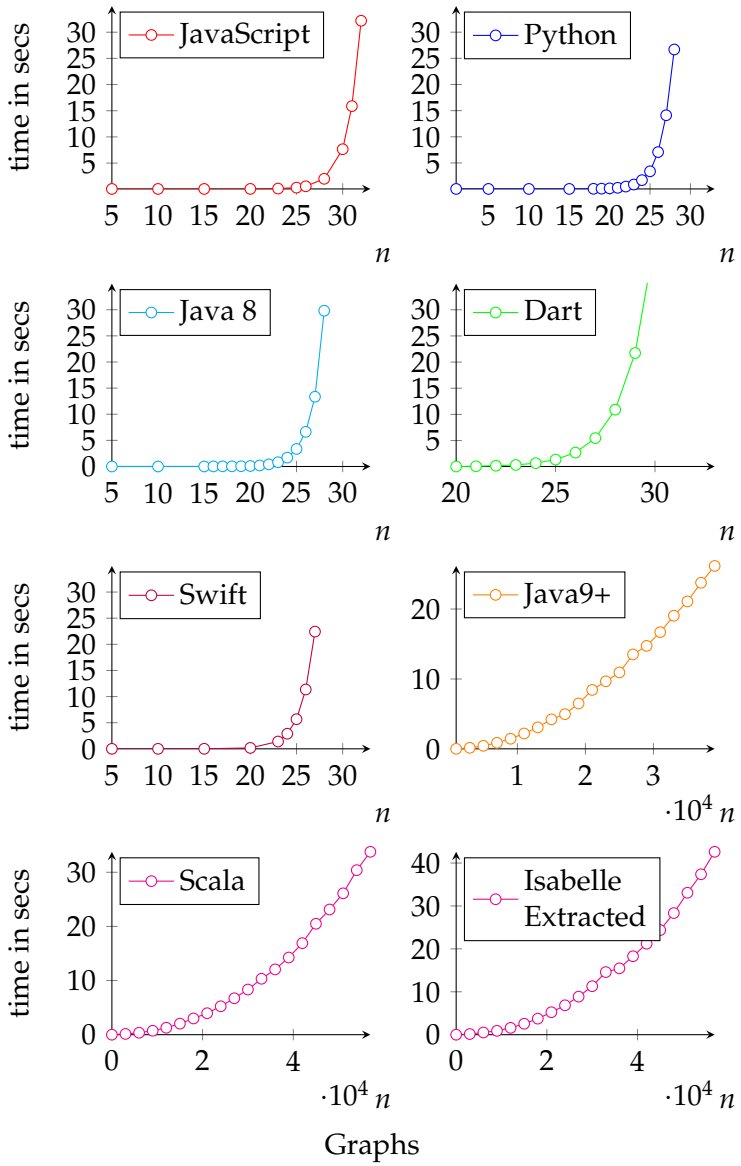


FIGURE 2.1: Graphs showing runtime for matching $(a^*)^* b$ with strings of the form $\underbrace{aa..a}_{n}$ in various existing regular expression libraries. The reason for their fast growth is that they do a depth-first-search using NFAs. If the string does not match, the regular expression matching engine starts to explore all possibilities. The last two graphs are for our implementation in Scala, one manual and one extracted from the verified lexer in Isabelle by *codegen*. Our lexer performs better in this case, and is formally verified. Despite being almost identical, the codegen-generated lexer is slower than the manually written version since the code synthesised by *codegen* does not use native integer or string types of the target language.

A more recent example is a global outage of all Cloudflare servers on 2 July 2019. The traffic Cloudflare services each day is more than Twitter, Amazon, Instagram, Apple, Bing and Wikipedia combined, yet it became completely unavailable for half an hour because of a poorly-written regular expression roughly of the form $^*.^* = .^*$ exhausted CPU resources 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] These problems with regular expressions are not isolated events that happen very rarely, but they occur actually often enough that they have a name: Regular-Expression-Denial-Of-Service (ReDoS) attacks. Davis et al. [29] detected more than 1000 evil regular expressions in Node.js, Python core libraries, npm and pypi. They therefore concluded that evil regular expressions are a real problem rather than just "a parlour trick".

The work in this thesis aims to address this issue with the help of formal proofs. We describe a lexing algorithm based on Brzozowski derivatives with verified correctness and a finiteness property for the size of derivatives (which are all done in Isabelle/HOL). Such properties are an important step in preventing catastrophic backtracking once and for all. We will give more details in the next sections on (i) why the slow cases in graph 2.1 can occur in traditional regular expression engines and (ii) why we choose our approach based on Brzozowski derivatives and formal proofs.

2.2 Preliminaries

Regular expressions and regular expression matchers have been studied for many years. Theoretical results in automata theory state that basic regular expression matching should be linear w.r.t the input. This assumes that the regular expression r was pre-processed and turned into a deterministic finite automaton (DFA) before matching [73]. By basic we mean textbook definitions such as the one below, involving only regular expressions for characters, alternatives, sequences, and Kleene stars:

$$r ::= c | r_1 + r_2 | r_1 \cdot r_2 | r^*$$

Modern regular expression matchers used by programmers, however, support much richer constructs, such as bounded repetitions, negations, and back-references. To differentiate, we use the word *regex* to refer to those expressions with richer constructs while reserving the term *regular expression* for the more traditional meaning in formal languages theory. We follow this convention in this thesis. In the future, we aim to support all the popular features of regexes, but for this work we mainly look at basic regular expressions and bounded repetitions.

Regexes come with a number of constructs that make it more convenient for programmers to write regular expressions. Depending on the types of constructs the task of matching and lexing with them will have different levels of complexity. Some of those constructs are syntactic sugars that are simply short hand notations that save the programmers a few keystrokes. These will not cause problems for regex libraries. For example the non-binary alternative involving three or more choices just means:

$$(a|b|c) \stackrel{means}{=} ((a + b) + c)$$

[2] https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/(Last accessed in 2022)

Similarly, the range operator is just a concise way of expressing an alternative of consecutive characters:

$$[0\ -9] \stackrel{means}{=} (0|1|\ldots|9)$$

for an alternative. The wildcard character '.' is used to refer to any single character,

$$. \stackrel{means}{=} [0-9a-zA-Z+-()*\&\ldots]$$

except the newline.

2.2.1 Bounded Repetitions

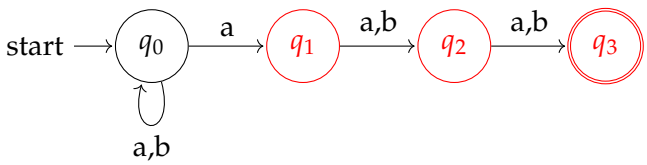
More interesting are bounded repetitions, which can make the regular expressions much more compact. Normally there are four kinds of bounded repetitions: $r^{\{n\}}$, $r^{\{\ldots m\}}$, $r^{\{n\ldots\}}$ and $r^{\{n\ldots m\}}$ (where n and m are constant natural numbers). Like the star regular expressions, the set of strings or language a bounded regular expression can match is defined using the power operation on sets:

$$\begin{array}{lcl} L\ r^{\{n\}} & \stackrel{\text{def}}{=} & (L\ r)^n \\ L\ r^{\{\ldots m\}} & \stackrel{\text{def}}{=} & \bigcup_{0 \le i \le m} .(L\ r)^i \\ L\ r^{\{n\ldots\}} & \stackrel{\text{def}}{=} & \bigcup_{n \le i} .(L\ r)^i \\ L\ r^{\{n\ldots m\}} & \stackrel{\text{def}}{=} & \bigcup_{n \le i \le m} .(L\ r)^i \end{array}$$

The attraction of bounded repetitions is that they can be used to avoid a size blow up: for example $r^{\{n\}}$ is a shorthand for the much longer regular expression:

$$\underbrace{r \ldots r}_{\text{n copies of r}} .$$

The problem with matching such bounded repetitions is that tools based on the classic notion of automata need to expand $r^{\{n\}}$ into n connected copies of the automaton for r. This leads to very inefficient matching algorithms or algorithms that consume large amounts of memory. Implementations using *DFA*s will in such situations either become very slow (for example Verbatim++ [33]) or run out of memory (for example *LEX* and *JFLEX*[3]) for large counters. A classic example for this phenomenon is the regular expression $(a+b)^*a(a+b)^n$ where the minimal DFA requires at least 2^{n+1} states. For example, when n is equal to 2, the corresponding *NFA* looks like:



[3]LEX and JFLEX are lexer generators in C and JAVA that generate *DFA*-based lexers. The user provides a set of regular expressions and configurations, and then gets an output program encoding a minimized *DFA* that can be compiled and run. When given the above countdown regular expression, a small n (say 20) would result in a program representing a DFA with millions of states.

and when turned into a DFA by the subset construction requires at least 2^3 states.[4]

Bounded repetitions are important because they tend to occur frequently in practical use, for example in the regex library RegExLib, in the rules library of Snort [72][5], as well as in XML Schema definitions (XSDs). According to Björklund et al [22], more than half of the XSDs they found on the Maven.org central repository have bounded regular expressions in them. Often the counters are quite large, with the largest being close to ten million. A smaller sample XSD they gave is:

```
<sequence minOccurs="0" maxOccurs="65535">
        <element name="TimeIncr" type="mpeg7:MediaIncrDurationType"/>
        <element name="MotionParams" type="float" minOccurs="2" maxOccurs="12"/>
</sequence>
```

This can be seen as the regex $(ab^{2...12})^{0...65535}$, where a and b are themselves regular expressions satisfying certain constraints (such as satisfying the floating point number format). It is therefore quite unsatisfying that some regular expressions matching libraries impose adhoc limits for bounded regular expressions: For example, in the regular expression matching library in the Go language the regular expression a^{1001} is not permitted, because no counter can be above 1000, and in the built-in Rust regular expression library expressions such as $a^{\{1000\}\{100\}\{5\}}$ give an error message for being too big [6]. As Becchi and Crawley [18] have pointed out, the reason for these restrictions is that they simulate a non-deterministic finite automata (NFA) with a breadth-first search. This way the number of active states could be equal to the counter number. When the counters are large, the memory requirement could become infeasible, and a regex engine like in Go will reject this pattern straight away.

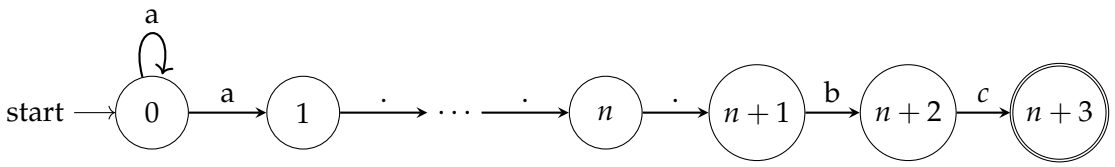


FIGURE 2.2: The example given by Becchi and Crawley that NFA simulation can consume large amounts of memory: $.^*a.^{\{n\}}bc$ matching strings of the form $aaa \ldots aaaabc$. When traversing in a breadth-first manner, all states from 0 till $n + 1$ will become active.

These problems can of course be solved in matching algorithms where automata go beyond the classic notion and for instance include explicit counters [81]. These

[4] The red states are "countdown states" which count 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 needs 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. The state q_3 is the last (accepting) state, requiring 0 more characters. Depending on the suffix of the input string up to the current read location, the states q_1 and q_2, q_3 may or may not be active. A *DFA* for such an *NFA* would contain at least 2^3 non-equivalent states that cannot be merged, because the subset construction during determinisation will generate all the elements in the power set $Pow\{q_1, q_2, q_3\}$. Generalizing this to regular expressions with larger bounded repetitions number, we have that regexes shaped like $r^*ar^{\{n\}}$ when converted to *DFA*s would require at least 2^{n+1} states, if r itself contains more than 1 string. This is to represent all different scenarios in which "countdown" states are active.

[5] Snort is a network intrusion detection (NID) tool for monitoring network traffic. The network security community curates a list of malicious patterns written as regexes, which is used by Snort's detection engine to match against network traffic for any hostile activities such as buffer overflow attacks.

[6] Try it out here: https://rustexp.lpil.uk

solutions can be quite efficient, with the ability to process gigabits of strings input per second even with large counters [18]. These practical solutions do not come with formal guarantees, and as pointed out by Kuklewicz [52], can be error-prone.

In the work reported in [**ITP2023**] and here, we add better support using derivatives for bounded regular expression $r^{\{n\}}$. Our results extend straightforwardly to repetitions with intervals such as $r^{\{n...m\}}$. The merit of Brzozowski derivatives (more on this later) on this problem is that it can be naturally extended to support bounded repetitions. Moreover these extensions are still made up of only small inductive datatypes and recursive functions, making it handy to deal with them in theorem provers. Finally, bounded regular expressions do not destroy our finite boundedness property, which we shall prove later on.

2.2.2 Back-References

The other way to simulate an *NFA* for matching is choosing a single transition each time, keeping all the other options in a queue or stack, and backtracking if that choice eventually fails. This method, often called a "depth-first-search", is efficient in many cases, but could end up with exponential run time. The backtracking method is employed in regex libraries that support *back-references*, for example in Java and Python.

Consider the following regular expression where the sequence operator is omitted for brevity:

$$r_1 r_2 r_3 r_4$$

In this example, one could label sub-expressions of interest by parenthesizing them and giving them a number in the order in which their opening parentheses appear. One possible way of parenthesizing and labelling is:

$$\underset{1}{(}r_1\underset{2}{(}r_2\underset{3}{(}r_3)\underset{4}{(}r_4)))$$

The sub-expressions $r_1r_2r_3r_4$, $r_1r_2r_3$, r_3 and r_4 are labelled by 1 to 4, and can be "referred back" by their respective numbers. To do so, one uses the syntax $\backslash i$ to denote that we want the sub-string of the input matched by the i-th sub-expression to appear again, exactly the same as it first appeared:

$$\ldots \underbrace{(\quad r_i)}_{\text{i-th lparen}} \ldots \underbrace{\backslash i}_{s_i \text{ which just matched } r_i} \ldots$$

Once the sub-string s_i for the sub-expression r_i has been fixed, there is no variability on what the back-reference label $\backslash i$ can be—it is tied to s_i. The overall string will look like $\ldots s_i \ldots s_i \ldots$ A concrete example for back-references is

$$(.^*)\backslash 1,$$

which matches strings that can be split into two identical halves, for example *foofoo*, *ww* and so on. Note that this is different from repeating the sub-expression verbatim like

$$(.^*)(.^*),$$

which does not impose any restrictions on what strings the second sub-expression $.^*$ might match. Another example for back-references is

$$(.)(.)\backslash 2\backslash 1$$

which matches four-character palindromes like $abba$, $x??x$ and so on.

Back-references are a regex construct that programmers find quite useful. According to Becchi and Crawley [18], 6% of Snort rules (up until 2008) use them. The most common use of back-references is to express well-formed html files, where back-references are convenient for matching opening and closing tags like

$$\langle html \rangle \ldots \langle /html \rangle$$

A regex describing such a format is

$$\langle (.^+) \rangle \ldots \langle / \backslash 1 \rangle$$

Despite being useful, the expressive power of regexes go beyond regular languages once back-references are included. In fact, they allow the regex construct to express languages that cannot be contained in context-free languages either. For example, the back-reference $(a^*)b\backslash 1b\backslash 1$ expresses the language $\{a^n b a^n b a^n \mid n \in \mathbb{N}\}$, which cannot be expressed by context-free grammars [24]. Such a language is contained in the context-sensitive hierarchy of formal languages. Also solving the matching problem involving back-references is known to be NP-complete [8]. Regex libraries supporting back-references such as PCRE [43] therefore have to revert to a depth-first search algorithm including backtracking. What is unexpected is that even in the cases not involving back-references, there is still a (non-negligible) chance they might backtrack super-linearly, as shown in the graphs in figure 2.1.

Summing up, we can categorise existing practical regex libraries into two kinds: (i) The ones with linear time guarantees like Go and Rust. The downside with them is that they impose restrictions on the regular expressions (not allowing back-references, bounded repetitions cannot exceed an ad hoc limit etc.). And (ii) those that allow large bounded regular expressions and back-references at the expense of using backtracking algorithms. They can potentially "grind to a halt" on some very simple cases, resulting ReDoS attacks if exposed to the internet.

The problems with both approaches are the motivation for us to look again at the regular expression matching problem. Another motivation is that regular expression matching algorithms that follow the POSIX standard often contain errors and bugs as we shall explain next.

2.3 Error-prone POSIX Implementations

Very often there are multiple ways of matching a string with a regular expression. In such cases the regular expressions matcher needs to disambiguate. The more widely used strategy is called POSIX, which roughly speaking always chooses the longest initial match. The POSIX strategy is widely adopted in many regular expression matchers because it is a natural strategy for lexers. However, many implementations (including the C libraries used by Linux and OS X distributions) contain bugs or do not meet the specification they claim to adhere to. Kuklewicz maintains a unit test repository which lists some problems with existing regular expression engines [52]. In some cases, they either fail to generate a result when there exists a match, or give results that are inconsistent with the POSIX standard. A concrete example is the regex:

$$(aba + ab + a)^* \text{and the string } ababa$$

The correct POSIX match for the above involves the entire string $ababa$, split into two Kleene star iterations, namely $[ab], [aba]$ at positions $[0, 2), [2, 5)$ respectively. But

feeding this example to the different engines listed at regex101 [7] [35]. yields only two incomplete matches: $[aba]$ at $[0,3)$ and a at $[4,5)$. Fowler [36] and Kuklewicz [52] commented that most regex libraries are not correctly implementing the central POSIX rule, called the maximum munch rule. Grathwohl [41] wrote:

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

People have recognised that the implementation complexity of POSIX rules also come from the specification being not very precise. The developers of the regexp package of Go [8] commented that

> *" The POSIX rule is computationally prohibitive and not even well-defined. "*

There are many informal summaries of this disambiguation strategy, which are often quite long and delicate. For example Kuklewicz [52] described the POSIX rule as (section 1, last paragraph):

- regular expressions (REs) take the leftmost starting match, and the longest match starting there earlier subpatterns have leftmost-longest priority over later subpatterns
- higher-level subpatterns have leftmost-longest priority over their component subpatterns
- REs have right associative concatenation which can be changed with parenthesis
- parenthesized subexpressions return the match from their last usage
- text of component subexpressions must be contained in the text of the higher-level subexpressions
- if "p" and "q" can never match the same text then "p|q" and "q|p" are equivalent, up to trivial renumbering of captured subexpressions
- if "p" in "p*" is used to capture non-empty text then additional repetitions of "p" will not capture an empty string

[7] regex101 is an online regular expression matcher which provides API for trying out regular expression engines of multiple popular programming languages like Java, Python, Go, etc.

[8] https://pkg.go.dev/regexp#pkg-overview

Chapter 3

Regular Expressions and POSIX Lexing-Preliminaries

This is a preliminary chapter which describes the results of Sulzmann and Lu [77] and Ausaf et al. [14].This chapter introduces the definitions and proofs related to *lexer*, one of the three lexing algorithms we are going to introduce. These basic definitions come partly from the AFP entry by Nipkow et al. [51] and partly from the AFP entry by Ausaf et al. [15]. Functions like *inj* formalisation results are not part of this PhD work, but the details are included to provide necessary context for our work on the correctness proof of *blexer_simp*, as we show in chapter 5 how the proofs break down when simplifications are applied.

In the coming section, the definitions of basic notions for regular languages and regular expressions are given. This is essentially a description in "English" the functions and datatypes used by Ausaf et al. [14] [13] in their formalisation in Isabelle/HOL. We include them as we build on their formalisation, and therefore inherently use these definitions.

As a general convention in this thesis, when we mention formally proven results from previous work, we call them *properties* instead of lemmas and omit their proofs unless they are key results. In that case, we call the key property a theorem and supply a high-level proof. These proofs are available from existing AFP entries by Ausaf et al. [15] or from their papers [14] [13], however we provide our perspective into how these proofs work intuitively. More importantly we show why they are no longer applicable for our correctness proof of *blexer_simp*.

3.1 Formal Specification of POSIX Matching and Brzozowski Derivatives

Brzozowski [23] first introduced the concept of a *derivative* of regular expression in 1964. The derivative of a regular expression r with respect to a character c, is written as $r\backslash c$. This operation tells us what r transforms into if we "chop" off a particular character c from all strings in the language of r (defined later as $L\ r$). Derivatives have the property that $s \in L\ (r\backslash c)$ if and only if $c :: s \in L\ r$ where :: stands for list prepending. With this property, derivatives can 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*). There are several mechanised proofs of this property in various theorem provers, for example one by Owens and Slind [66] in HOL4, another one by Krauss and Nipkow [62] in Isabelle/HOL, and yet another in Coq by Coquand and Siles [27].

In addition, one can extend derivatives to bounded repetitions relatively straightforwardly. For example, the derivative for this can be defined as:

$$r^{\{n\}}\backslash c \quad \stackrel{\text{def}}{=} \quad r\backslash c \cdot r^{\{n-1\}} (\text{when} n > 0)$$

Experimental results suggest that unlike DFA-based solutions for bounded regular expressions, derivatives can cope large counters quite well.

There have also been extensions of derivatives to other regex constructs. For example, Owens et al include the derivatives for the *NOT* regular expression, which is able to concisely express C-style comments of the form $/ * \ldots * /$ (see [66]). Another extension for derivatives is regular expressions with look-aheads, done by Miyazaki and Minamide [59].

Given the above definitions and properties of Brzozowski derivatives, one quickly realises their potential in generating a formally verified algorithm for lexing: the clauses and property can be easily expressed in a functional programming language or converted to theorem prover code, with great ease. Perhaps this is the reason why derivatives have sparked quite a bit of interest in the functional programming and theorem prover communities in the last fifteen or so years ([9], [56], [20], [87] and [27] to name a few), despite being buried in the "sands of time" [66] after they were first published by Brzozowski.

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). Second, derivative-based matchers need to be more efficient in terms of the sizes of derivatives. Elegant and beautiful as many implementations are, they can be still quite slow. For example, Sulzmann and Lu claim a linear running time of their proposed algorithm, but that was falsified by our experiments. 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 [32] lexer, which calculates POSIX matches and is based on derivatives. They formalized the correctness of the lexer, but not their complexity result. In the performance evaluation section, they analyzed the run time of matching a with the string

$$\underbrace{a \ldots a}_{\text{n a's}}.$$

They concluded that the algorithm is quadratic in terms of the length of the input string. When we tried out their extracted OCaml code with the example $(a + aa)^*$, the time it took to match a string of 40 a's was approximately 5 minutes.

3.1.1 Sulzmann and Lu's Algorithm

Sulzmann and Lu [77] overcame the first problem with the yes/no answer 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. This injection function in a sense undoes the "damage" of the derivatives chopping off characters. In earlier work, Ausaf et al provided the formal specification of what POSIX matching means and proved in Isabelle/HOL the correctness of this extended algorithm accordingly [14].

The version of the algorithm proven correct suffers however heavily from a second difficulty, where 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

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

where after around 35 steps we usually run out of memory on a typical computer. Clearly, the notation involving $\mathbf{0}$s and $\mathbf{1}$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 the work by Ausaf et al. to preserve the correctness of Sulzmann and Lu's algorithm [14], 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.

Therefore we want to look in this thesis at a second algorithm by Sulzmann and Lu where they overcame this "growth problem" [77]. In this version, POSIX values are represented as bit sequences and such sequences are incrementally generated when derivatives are calculated. The compact representation of bit sequences 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 bit-coded *incremental parsing method* (that is the algorithm to be formalised in this dissertation)

> *"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." [77, Page 14]*

Ausaf and Urban made some initial progress towards the full correctness proof but still had to leave out the optimisation Sulzmann and Lu proposed. Ausaf wrote [13],

> *"The next step would be to implement a more aggressive simplification procedure on annotated regular expressions and then prove the corresponding algorithm generates the same values as blexer. Alas due to time constraints we are unable to do so here."*

This thesis implements the aggressive simplifications envisioned by Ausaf and Urban, together with a formal proof of the correctness of those simplifications.

One of the most recent work in the context of lexing is the Verbatim lexer by Egolf, Lasser and Fisher [32]. This is relevant work for us and we will compare it later with our derivative-based matcher we are going to present. There is also some newer work called Verbatim++ [33], which does not use derivatives, but deterministic finite automaton instead. We will also study this work in a later section.

3.2 Basic Concepts

Formal language theory usually starts with an alphabet denoting a set of characters. Here we use the datatype of characters from Isabelle, which roughly corresponds to the ASCII characters. In what follows, we shall leave the information about the alphabet implicit. Strings are defined as a list characters, and we use the Scala notation for list Cons operation: appending a character to the front of a list is written as $c :: s$. For brevity, a singleton list is sometimes written as $[c]$. Strings can be concatenated to form longer strings in the same way we concatenate two lists, which we shall write as $s_1@s_2$. We omit the precise recursive definition here. We overload the @ operator for language concatenation as well, for example $A@B$ where A and B are two sets. The power of a language is defined recursively, using the language concatenation operator @:

$$\begin{array}{lcl} A^0 & \stackrel{\text{def}}{=} & \{[]\} \\ A^{n+1} & \stackrel{\text{def}}{=} & A@A^n \end{array}$$

The union of all powers of a language can be used to define the Kleene star operator:

$$A* \stackrel{\text{def}}{=} \bigcup_{i \geq 0} A^i$$

However, to obtain a more convenient induction principle in Isabelle/HOL, we reuse the definitions by the AFP entry [15] where they define the Kleene star as an inductive set, we are aware of another AFP entry on Kleene algebra, but for convenice we do not refactor the Isabelle code base by Ausaf et al.

$$\frac{}{[] \in A*} \qquad \frac{s_1 \in A \quad s_2 \in A*}{s_1@s_2 \in A*}$$

We also define an operation of "chopping off" a character from a language, which we call *Der*, meaning *Derivative* (for a language):

$$Der\ c\ A \stackrel{\text{def}}{=} \{s \mid c :: s \in A\}$$

This can be generalised to "chopping off" a string from all strings within a set A, namely:

$$Ders\ s\ A \stackrel{\text{def}}{=} \{s' \mid s@s' \in A\}$$

which is essentially the left quotient $A\backslash L$ of A against the singleton language with $L = \{s\}$. However, for our purposes here, the *Ders* definition with a single string is sufficient.

The reason for defining derivatives is that they provide another approach to test membership of a string in a set of strings. For example, to test whether the string *bar* is contained in the set $\{foo, bar, brak\}$, one can take derivative of the set with respect to the string *bar*:

$$\begin{array}{lcl} S = \{foo, bar, brak\} & \stackrel{Der\ b}{\rightarrow} & \{ar, rak\} \\ & \stackrel{Der\ a}{\rightarrow} & \{r\} \\ & \stackrel{Der\ r}{\rightarrow} & \{[]\} \end{array}$$

and in the end, test whether the set contains the empty string.[1] This idea originally came up already in the paper by Brzozowski [23], and there is an AFP entry formalising this [**UrbanAFP**]. Usually language derivatives are defined and formalised alongside the definition of regular expression derivatives to indicate that the semantics of regular expression derivatives are well-defined, namely $L\ (r\backslash c) = Der\ c\ (L\ r)$. We are going to introduce this property in more detail later.

In general, if we have a language S, then we can test whether s is in S by testing whether $[] \in S\backslash s$. In the previous $S = \{foo, bar, brak\}$ example, we can tell that *bar* is in S because the empty string is in *Ders bar S*. We list some notable properties about language derivatives as they are conducive of the definition of regular expression derivatives:

- $Der\ c\ (A@B) = \begin{cases} ((Der\ c\ A)\ @\ B) \cup (Der\ c\ B), & \text{if } [] \in A \\ (Der\ c\ A)\ @\ B, & \text{otherwise} \end{cases}$
- $Der\ c\ (A*) = (Der\ c A)@(A*)$

We omit the proofs, as they can be found in the AFP entry by Ausaf et al. [13]. Some examples for language derivatives:

$$\begin{array}{lcl} Der\ a\ \{ab\}* & = & (Der\ a\ \{ab\})@\{ab\}* \\ & = & \{b\}@(\{ab\}*) \\ & = & \{b,\ bab,\ bababab\ldots\} \\ Der\ a\ (\{ab,\ a,\ []\}@\{b,\ a\}) & = & (Der\ a\ \{ab,\ a,\ []\})@\{b,\ a\} \cup (Der\ a\ \{b,\ a\}) \\ & = & \{b,\ []\}@\{b,\ a\} \cup \{[]\} \\ & = & \{bb,\ ba,\ b,\ a,\ []\} \end{array}$$

The clever idea of Brzozowski was to find the counterpart of *Der* for regular expressions. To introduce them, we need to first give definitions for regular expressions, which we shall do next.

3.2.1 Regular Expressions and Derivatives

The *basic regular expressions* are defined inductively by the following grammar:*added types for regex, but looks slightly weird*

$$\begin{array}{lcl} r & :: & rexp \\ r & ::= & \mathbf{0} \mid \mathbf{1} \mid c \mid r_1 \cdot r_2 \mid r_1 + r_2 \mid r^* \end{array}$$

We call them basic because we will introduce additional constructors in later chapters, such as negation and bounded repetitions. We use $\mathbf{0}$ for the regular expression that matches no string, and $\mathbf{1}$ for the regular expression that matches only the empty string.[2] The sequence regular expression is written $r_1 \cdot r_2$ and sometimes we omit the dot if it is clear which regular expression is meant; the alternative is written $r_1 + r_2$. The *language* or meaning of a regular expression is defined recursively as a set of strings:

[1] We use the infix notation $A\backslash c$ instead of $Der\ c\ A$ for brevity, as it will always be clear from the context that we are operating on languages rather than regular expressions.

[2] Some authors also use ϕ and ϵ for $\mathbf{0}$ and $\mathbf{1}$ but we prefer this notation.

$$
\begin{array}{lcl}
L & :: & \mathit{rexp} \Rightarrow \mathit{string\ set} \\
L\ \mathbf{0} & \stackrel{\text{def}}{=} & \varnothing \\
L\ \mathbf{1} & \stackrel{\text{def}}{=} & \{[]\} \\
L\ c & \stackrel{\text{def}}{=} & \{[c]\} \\
L\ (r_1 + r_2) & \stackrel{\text{def}}{=} & L\ r_1 \cup L\ r_2 \\
L\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L\ r_1 @ L\ r_2 \\
L\ (r^*) & \stackrel{\text{def}}{=} & (L\ r)*
\end{array}
$$

Brzozowski noticed that *Der* can be "mirrored" on regular expressions which he calls the derivative of a regular expression r with respect to a character c, written $r\backslash c$. This infix operator takes regular expression r as input and a character as a right operand. Here is its recursive definition:

$$
\begin{array}{lcl}
\mathbf{0}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathbf{1}\backslash c & \stackrel{\text{def}}{=} & \mathbf{0} \\
d\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ c = d\ \mathit{then}\ \mathbf{1}\ \mathit{else}\ \mathbf{0} \\
(r_1 + r_2)\backslash c & \stackrel{\text{def}}{=} & r_1\backslash c\ +\ r_2\backslash c \\
(r_1 \cdot r_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ [] \in L(r_1) \\
 & & \mathit{then}\ (r_1\backslash c) \cdot r_2\ +\ r_2\backslash c \\
 & & \mathit{else}\ (r_1\backslash c) \cdot r_2 \\
(r^*)\backslash c & \stackrel{\text{def}}{=} & (r\backslash c) \cdot r^*
\end{array}
$$

Regular expression derivatives are defined such that the language of the derivative result coincides with the language of the original regular expression being taken derivative with respect to the same characters. For example, the two previous examples have their corresponding calculations on regular expressions:

$$
\begin{array}{lcl}
\mathit{Der}\ a\ \{ab\}* & = & (\mathit{Der}\ a\ \{ab\})@\{ab\}* \\
 & = & \{b\}@(\{ab\}*) \\
 & = & \{b,\ bab,\ babab\ldots\} \\
(ab)^*\backslash a & = & (ab)\backslash a \cdot (ab)^* \\
 & = & (\mathbf{1}b) \cdot (ab)^* \\
\mathit{Der}\ a\ (\{ab,\ a,\ []\}@\{b,\ a\}) & = & (\mathit{Der}\ a\ \{ab,\ a,\ []\})@\{b,\ a\} \cup (\mathit{Der}\ a\ \{b,\ a\}) \\
 & = & \{b,\ []\}@\{b,\ a\} \cup \{[]\} \\
 & = & \{bb,\ ba,\ b,\ a,\ []\} \\
(ab + a + []) \cdot (b + a)\backslash a & = & (\mathbf{1}b + \mathbf{1} + \mathbf{0}) \cdot (b + a) + (\mathbf{0} + \mathbf{1})
\end{array}
$$

The reader might notice that unlike *Der*, we did not simplify alongside the calculations. A matcher does allow simplifications, but the lexer we are going to introduce in this chapter. We will show how to enable simplifications whilst maintaining the lexical value in chapter 5.

The most involved cases are the sequence case and the star case. The sequence case says that if the first regular expression contains an empty string, then the second component of the sequence needs to be considered, as its derivative will contribute to the result of this derivative. This is isomorphic to *Der* on language concatenations. Similarly, the derivative of the star regular expression r^* unwraps one iteration of r, turns it into $r\backslash c$, and attaches the original r^* after $r\backslash c$, so that we can further unfold it as many times as needed: Again, the structure is the same as the language derivative of the Kleene star. In the above definition of $(r_1 \cdot r_2)\backslash c$, the *if* clause's boolean condition $[] \in L(r_1)$ needs to be recursively computed. This function is usually called *nullable*:

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

The $\mathbf{0}$ regular expression does not contain any string and therefore is not *nullable*. $\mathbf{1}$ is *nullable* by definition. The character regular expression c corresponds to the singleton set $\{c\}$, and therefore does not contain the empty string. The alternative regular expression is nullable if at least one of its children is nullable. The sequence regular expression would require both children to have the empty string to compose an empty string, and the Kleene star is always nullable because it naturally contains the empty string.

One can overload the derivative to strings and build a matcher on top of string derivatives:

$$\begin{array}{lcl} r\backslash(c::s) & \stackrel{\text{def}}{=} & (r\backslash c)\backslash s \\ r\backslash[\,] & \stackrel{\text{def}}{=} & r \\ \mathit{match}\ r\ s\ ::\ \text{"}\mathit{rexp} \Rightarrow \mathit{string} \Rightarrow \mathit{bool}\text{"} & \stackrel{\text{def}}{=} & \mathit{nullable}(r\backslash s) \end{array}$$

This matcher has already been described by Brzozowski [23]. Here are some well-known but important properties about regular expression derivatives and matching:

- $\mathit{Der}\ c\ L(r) = L(r\backslash c)$
- $c::s \in L(r)$ *iff* $s \in L(r\backslash c)$.
- $\mathit{match}\ s\ r = \mathit{true}$ *iff* $s \in L(r)$

These are the correctness theorems of derivative-based matchers, and variants of them have been described in various work on regular expressions and formal proofs (e.g. [66], [50], [27], [71]. and [33]).
Assuming the string is given as a sequence of characters, say $c_0 c_1 \ldots c_n$, this algorithm, presented graphically, is as follows:

$$r_0 \xrightarrow{\backslash c_0} r_1 \xrightarrow{\backslash c_1} r_2 \dashrightarrow r_n \xrightarrow{\mathit{nullable?}} \text{true/false}$$

Here are some example computations for the matcher:

- $(a^*)\backslash a = \mathbf{1} \cdot a^*$ is nullable $\rightarrow \mathit{true}$
- $(ab + c)\backslash ab = (\mathbf{1} \cdot b + \mathbf{0})\backslash b = (\mathbf{0} \cdot b + \mathbf{1}) + \mathbf{0}$ is nullable $\rightarrow \mathit{true}$
- $(a + ab) \cdot (b + \mathbf{1})\backslash ab = ((\mathbf{1} + \mathbf{1}b) \cdot (b + \mathbf{1}))\backslash b = (\mathbf{0} + (\mathbf{0}b + \mathbf{1})) \cdot (b + \mathbf{1}) + (\mathbf{1} + \mathbf{0})$ is nullable $\rightarrow \mathit{true}$
- $(cc) * \backslash c = (\mathbf{1}c) \cdot (cc)^*$ is not nullable $\rightarrow \mathit{false}$, this means that $c \notin L\ (cc)^*$

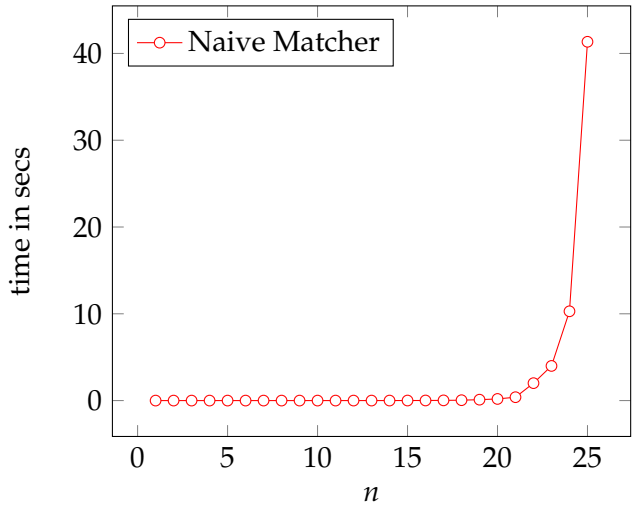


FIGURE 3.1: Matching the regular expression $(a^*)^*b$ against strings of the form $\underbrace{aa\ldots a}_{\text{n as}}$ using Brzozowski's original algorithm

- $((aba + ab) + a)^*\backslash aba = (((\mathbf{1}ba + \mathbf{1}b) + \mathbf{1}) \cdot ((aba + ab) + a)^*)\backslash ba = ((((\mathbf{1}a + \mathbf{0}ba) + (\mathbf{0}b + \mathbf{1})) + \mathbf{0}) \cdot ((aba + ab) + a)^* + ((\mathbf{0}ba + \mathbf{0}b) + \mathbf{0}) \cdot ((aba + ab) + a)^*)\backslash a = ((((((\mathbf{0}a + \mathbf{1}) + \mathbf{0}ba) + (\mathbf{0}b + \mathbf{0})) + \mathbf{0}) \cdot ((aba + ab) + a)^* + (((\mathbf{1}ba + \mathbf{1}b) + \mathbf{1}) \cdot ((aba + ab) + a)^*)) + (((\mathbf{1}a + \mathbf{0}ba) + (\mathbf{0}b + \mathbf{1})) + \mathbf{0}) \cdot ((aba + ab) + a)^*$ is nullable $\rightarrow$ *true*

As can be seen from the last example, the size of the derivatives can grow exponentially quickly. If one implements a matcher using the above recursive functions naively without any changes, the program can be as slow as backtracking matchers, as shown in 3.1. Note that both axes are in logarithmic scale. Around two dozen characters this algorithm already "explodes" with the regular expression $(a^*)^*b$. To improve this situation, we need to introduce simplification rules for the intermediate results, such as $r + r \rightarrow r$ or $\mathbf{1} \cdot r \rightarrow r$, and make sure those rules do not change the language of the regular expression. One simplification function that achieves these requirements is given below (see Ausaf et al. [14]):

$$
\begin{array}{lcl}
simp & :: & rexp \Rightarrow rexp \\
simp\ r_1 \cdot r_2 & \stackrel{\text{def}}{=} & (simp\ r_1, simp\ r_2)\ match \\
 & & \quad case\ (\mathbf{0}, \_) \Rightarrow \mathbf{0} \\
 & & \quad case\ (\_, \mathbf{0}) \Rightarrow \mathbf{0} \\
 & & \quad case\ (\mathbf{1}, r_2') \Rightarrow r_2' \\
 & & \quad case\ (r_1', \mathbf{1}) \Rightarrow r_1' \\
 & & \quad case\ (r_1', r_2') \Rightarrow r_1' \cdot r_2' \\
simp\ r_1 + r_2 & \stackrel{\text{def}}{=} & (simp\ r_1, simp\ r_2) match \\
 & & \quad case\ (\mathbf{0}, r_2') \Rightarrow r_2' \\
 & & \quad case\ (r_1', \mathbf{0}) \Rightarrow r_1' \\
 & & \quad case\ (r_1', r_2') \Rightarrow r_1' + r_2' \\
simp\ r & \stackrel{\text{def}}{=} & r \qquad (otherwise)
\end{array}
$$

If we repeatedly apply this simplification function during the matching algorithm, we have a matcher with simplification:

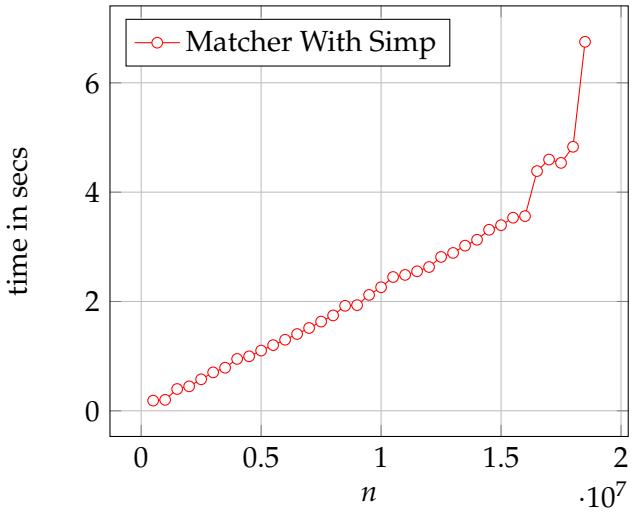


FIGURE 3.2: $(a^*)^*b$ against $\underbrace{aa\ldots a}_{\text{n as}}$ Using $matcher_{simp}$

$$
\begin{array}{lcl}
ders\_simp\ []\ r & \stackrel{\text{def}}{=} & r \\
ders\_simp\ c :: cs\ r & \stackrel{\text{def}}{=} & ders\_simp\ cs\ (simp\ (r\backslash c)) \\
matcher_{simp}\ s\ r & \stackrel{\text{def}}{=} & nullable\ (ders\_simp\ s\ r)
\end{array}
$$

The running time of *ders_simp* on the same example of Figure 3.1 is now "tame" in terms of the length of inputs, as shown in Figure 3.2.

So far, the story is use Brzozowski derivatives and simplify as much as possible, and at the end test whether the empty string is recognised by the final derivative. But what if we want to do lexing instead of just getting a true/false answer? Sulzmann and Lu [77] first came up with a nice and elegant (arguably as beautiful as the definition of the Brzozowski derivative) solution for this. For the same reason described at the beginning of this chapter, we introduce the formal semantics of *POSIX* lexing by Ausaf et al.[14], followed by the first lexing algorithm by Sulzmanna and Lu [77] that produces the output conforming to the *POSIX* standard. In what follows we choose to use the Isabelle-style notation for function and datatype constructor applications, where the parameters of a function are not enclosed inside a pair of parentheses (e.g. $f\ x\ y$ instead of $f(x,\ y)$). This is mainly to make the text visually more concise.

3.3 Values and the Lexing Algorithm by Sulzmann and Lu

In this section, we present a two-phase regular expression lexing algorithm by Sulzmann and Lu [79]. We refer to it as *lexer*. The first phase of *lexer* takes successive derivatives with respect to the input string, and the second phase does the reverse, *injecting* back characters, in the meantime constructing a lexing result. We will introduce the injection phase in detail slightly later, but as a preliminary we have to first define the datatype for lexing results, called *value* or sometimes also *lexical value*. Note that these definitions can be found in the AFP entry by Ausaf et al. [15], and we introduce them because we directly build on these formalisations. 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^*		$\mid$	$Stars\ [v_1, \ldots v_n]$

A value has an underlying string, which can be calculated by the "flatten" function $|\_|$:

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

Sulzmann and Lu used a binary predicate, written $\vdash v : r$, to indicate that a value v could be generated from a lexing algorithm with input r. They call it the value inhabitation relation, defined by the rules.

$$
\frac{}{\vdash Char\ c : \mathbf{c}} \qquad \frac{}{\vdash Empty : \mathbf{1}} \qquad \frac{\vdash v_1 : r_1 \quad \vdash v_2 : r_2}{\vdash Seq\ v_1,\ v_2 : (r_1 \cdot r_2)}
$$

$$
\frac{\vdash v_1 : r_1}{\vdash Left\ v_1 : r_1 + r_2} \qquad \frac{\vdash v_2 : r_2}{\vdash Right\ v_2 : r_1 + r_2} \qquad \frac{\forall v \in vs.\ \vdash v : r \wedge |v| \neq []}{\vdash Stars\ vs : r^*}
$$

The condition $|v| \neq []$ in the premise of star's rule is to make sure that for a given pair of regular expression r and string s, the number of values satisfying $|v| = s$ and $\vdash v : r$ is finite. This additional condition was imposed by Ausaf and Urban to make their proofs easier. Given a string and a regular expression, there can be multiple values for it. For example, both $\vdash Seq(Left\ ab)(Right\ c) : (ab + a)(bc + c)$ and $\vdash Seq(Right\ a)(Left\ bc) : (ab + a)(bc + c)$ hold and the values both flatten to abc. Lexers therefore have to disambiguate and choose only one of the values to be generated. *POSIX* is one of the disambiguation strategies that is widely adopted.

Ausaf et al. [14] formalised the property as a ternary relation. The *POSIX* value v for a regular expression r and string s, denoted as $(s, r) \to v$, can be specified in the following rules[3]:

The above *POSIX* rules follow the intuition described below:

- (Left Priority)
  Match the leftmost regular expression when multiple options for matching are available. See P+L and P+R where in P+R s cannot be in the language of $L\ r_1$.

- (Maximum munch)
  Always match a subpart as much as possible before proceeding to the next part

[3]The names of the rules are used as they were originally given in [14].

$$\text{P1}\quad \frac{}{([], \mathbf{1}) \to \mathit{Empty}} \qquad \text{PC}\quad \frac{}{([c], c) \to \mathit{Char}\ c} \qquad \text{P+L}\quad \frac{(s, r_1) \to v_1}{(s, r_1 + r_2) \to \mathit{Left}\ v_1}$$

$$\text{P+R}\quad \frac{(s, r_2) \to v_2 \qquad s \notin L\ r_1}{(s, r_1 + r_2) \to \mathit{Right}\ v_2}$$

$$\text{PS}\quad \frac{(s_1, v_1) \to r_1 \qquad (s_2, v_2) \to r_2 \qquad \nexists s_3\ s_4.s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L\ r_1 \wedge s_4 \in L\ r_2}{(s_1 @ s_2, r_1 \cdot r_2) \to \mathit{Seq}\ v_1\ v_2}$$

$$\text{P[]}\quad \frac{}{([], r^*) \to \mathit{Stars}([])}$$

$$\text{P*}\quad \frac{(s_1, v) \to v \qquad (s_2, r^*) \to \mathit{Stars}\ vs \qquad |v| \neq [] \qquad \nexists s_3\ s_4.s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L\ r \wedge s_4 \in L\ r^*}{(s_1 @ s_2, r^*) \to \mathit{Stars}\ (v :: vs)}$$

FIGURE 3.3: The inductive POSIX rules given by Ausaf et al. [14]. This ternary relation, written $(s, r) \to v$, formalises the POSIX constraints on the value v given a string s and regular expression r.

of the string. For example, when the string s matches $r_{part1} \cdot r_{part2}$, and we have two ways s can be split: Then the split that matches a longer string for the first part r_{part1} is preferred by this maximum munch rule. The side-condition

$$\nexists s_3\ s_4. s_3 \neq [] \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L\ r_1 \wedge s_4 \in L\ r_2$$

in PS causes this.

These disambiguation strategies can be quite practical. For instance, when lexing a code snippet

$$\mathit{iffoo} = 3$$

using a regular expression for keywords and identifiers:

$$r_{keyword} + r_{identifier}.$$

If we want *iffoo* to be recognized as an identifier where identifiers are defined as usual (letters followed by letters, numbers or underscores), then a match with a keyword (if) followed by an identifier (foo) would be incorrect. POSIX lexing generates what is included by lexing.

Here are some important properties about POSIX values, they have been proven by Ausaf et al. [14]:

- A POSIX value for regular expression r is inhabited by r. $(r, s) \rightarrow v \implies \vdash v : r$
- Given the same regular expression r and string s, one can always uniquely determine the *POSIX* value for it: $\mathit{if}\ (s, r) \rightarrow v_1 \wedge (s, r) \rightarrow v_2 \quad \mathit{then}\ v_1 = v_2$

We omit the proofs here (as they can be found in the corresponding AFP entry [15]), but provide some examples. For the first property, we have:

- $(((aba + ab) + a)^*, aba) \rightarrow Stars[Left\ Left\ aba]$ implies $\vdash Stars[Left\ Left\ aba] : ((aba + ab) + a)^*$ (aba is not fully expanded into sequence value contructors to avoid redundancy, same for below).
- Also we have $(((aba + ab) + a)^*, ababa) \rightarrow Stars[Left\ Right\ ab, Left\ Left\ aba]$ and $\vdash Stars[Left\ Right\ ab, Left\ Left\ aba] : (((aba + ab) + a)^*, ababa)$ holds.

For the second property, the examples of some POSIX values:

- For $(aa + a)^*$, we have $((aa + a)^*, aaa) \rightarrow Stars[Left\ \ aa, Righta]$, and $((aa + a^*, aaaa), aaaa) \rightarrow Stars[Left\ aa, Left\ aa]$ holds.
- $((ab + a) \cdot (b + \mathbf{1}), ab) \rightarrow Seq\ (Left\ ab)$ holds. $\vdash Seq\ (Right\ a)\ (Left\ b) : (ab + a) \cdot (b + \mathbf{1})$ holds, but $Seq\ (Right\ a)\ (Left\ b)$ is not a POSIX value.

We have now given the definition of what a POSIX value is and provided formally proven results that it is unique. The problem is to generate such a value in a lexing algorithm using derivatives.

3.3.1 Sulzmann and Lu's Injection-based Lexing Algorithm

Sulzmann and Lu extended Brzozowski's derivative-based matching to a lexing algorithm by a second phase after the initial phase of successive derivatives of r with respect to s. This second phase generates a POSIX value if the regular expression matches the string. The algorithm uses two functions called *inj* and *mkeps*. The

value produced by *mkeps* tells us how the empty string is matched by $r\backslash s$. Then *inj* is called to incrementally build on this value to get finally how r matched s. The definition of *mkeps* ([77]) is

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

The function prefers the left child r_1 of $r_1 + r_2$ to match an empty string if there is a choice. When there is a star to match the empty string, we give the *Stars* constructor an empty list, meaning no iteration is taken. The following property holds:

Property 1.

- *The result of mkeps on a nullable r is a POSIX value for r and the empty string:* $\mathit{nullable}\ r \implies (r, []) \to (\mathit{mkeps}\ r)$

The property has been proven by Ausaf et al., see [15] for details. Here are some examples of how *mkeps* picks up the POSIX value, note how the left alternative is preferred over the right when nullable:

- $(\mathbf{0} + \mathbf{1}, []) \to \mathit{Right}\ \mathit{Empty}$
- $((\mathbf{0} + (\mathbf{0}b + \mathbf{1})) \cdot (b + \mathbf{1}) + (\mathbf{1} + \mathbf{0}), []) \to \mathit{Left}\ (\mathit{Seq}\ (\mathit{Right}\ (\mathit{Right}\ \mathit{Empty}))\ (\mathit{Right}\ \mathit{Empty}))$
- $(\mathbf{1} \cdot a^*, []) \to \mathit{Seq}\ \mathit{Empty}\ (\mathit{Stars}\ [])$

After the *mkeps*-call, Sulzmann and Lu inject back the characters one by one in reverse order as they were chopped off in the derivative phase. The function for this is called *inj*. The definition of *inj* is as follows (Ausaf et al. called this *injval* in their AFP entry [15]):

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

Attentive readers will notice that the pattern matching is not exhaustive, and the output is set to *undefined* when *inj* is called on malformed input. Fortunately *lexer* will never use *inj* in such a way that it gives the *undefined* result. This can be proven in theorem 1. The function *inj* operates on values, unlike $\backslash$ which operates on regular expressions. If one just reverses a derivative like $r\backslash c$, then only the original regular expression r is returned. But we do not want r, what we want is how r matched the string cs, where s matched $r\backslash c$. Therefore the "injection" operations needs to be defined on *values* instead. Also, injecting characters into values poses a question: how does one know where to inject the character into? Take the last example above, let

$$r\backslash c = \mathbf{1} \cdot a^*, s = [], \text{and } c = a.$$

In the absence of the original regular expression r it is not clear which place in

$$Seq\ Empty\ (Stars\ [])$$

should one inject a into. This requires some additional information, namely what the regular expression was before the derivative took place. In this example, we know that the original regular expression was a^*, and the derivative is of form $(a\backslash a) \cdot a^*$, and therefore the place to "invert" the derivative is the front part of the sequence value which is *Empty*. This corresponds to the Star clause in *inj*'s definition. Here are a few more examples which cover some of the clauses of *inj*:

- $((a + b)\backslash b = (\mathbf{0} + \mathbf{1}), []) \rightarrow Right\ Empty \implies$
  $((a + b), b) \rightarrow inj\ (a + b)\ b\ (Right\ Empty) = Right\ b$ (Right clause of alternative used)
- $(\mathbf{1} \cdot a^*, []) \rightarrow Seq\ Empty\ Stars\ []$
  $(a^*, a) \rightarrow inj\ a^*\ a\ (Seq\ Empty\ (Stars\ []))Stars\ [inj\ a\ a\ Empty] = Stars\ [a]$ (Star clause used)
- $(((\mathbf{0} + \mathbf{1}) \cdot \mathbf{1}) + \mathbf{0}, []) \rightarrow Left\ (Seq\ (Right\ a)\ Empty)$
  $((\mathbf{1} + a) \cdot \mathbf{1}, a) \rightarrow Seq\ (Right\ a)\ Empty$ (Left clause of sequence used)

Given r, c, s and the POSIX value v such that $(r\backslash c, s) \rightarrow v$, one can always construct a value $inj\ r\ c\ v$ which preserves POSIXness. This can be proven by a case analysis on r and $r\backslash c$, where each case corresponds to a clause of *inj*. We omit the proof and refer the curious readers to the AFP entry [15] or the paper [14] for higher-level proof content.

Property 2.

If $(r\backslash c, s) \rightarrow v$, *then* $(r, c :: s) \rightarrow inj\ r\ c\ v$.

Putting $\backslash$, *mkeps* and *inj* together, Sulzmann and Lu obtained the following lexing algorithm:

$$\begin{array}{lcl} lexer\ r\ [] & = & if\ (nullable\ r)\ then\ Some(mkeps\ r)\ else\ None \\ lexer\ r\ c :: s & = & case\ (lexer\ (r\backslash c)\ s)\ of \\ & & \quad None \Rightarrow None \\ & & \quad \mid Some\ v \Rightarrow Some\ (inj\ r\ c\ v) \end{array}$$

Pictorially, this can be represented as follows:

$$\begin{array}{ccccccc} r_0 & \dashrightarrow & r_i & \xrightarrow{\backslash c_i} & r_{i+1} & \dashrightarrow & r_n \\ & & & & & & \downarrow mkeps \\ v_0 & \dashleftarrow & v_i & \xleftarrow[inj_{r_i} c_i]{} & v_{i+1} & \dashleftarrow & v_n \end{array}$$

In the diagram above, v_i stands for the (POSIX) value for how the regular expression r_i matches the string s_i consisting of the last $n - i$ characters of s (i.e. $s_i = c_i \ldots c_{n-1}$) from the previous lexical value v_{i+1}. After injecting back s's characters, we get the lexical value for how r_0 matches s. Here are a few concrete examples for *lexer*:

$$r_0 = (aa)^* \dashrightarrow r_1 = (\mathbf{1}a)\cdot(aa)^* \xrightarrow{\backslash c_1 = a} r_2 = (\mathbf{0}a + \mathbf{1})\cdot(aa)^*$$
$$\downarrow mkeps$$
$$v_0 = Stars\ [aa] \dashleftarrow v_1 = Seq\ (Seq\ Empty\ a)\ (Stars\ []) \xleftarrow{inj_{r_1}\, c_1} v_2 = Seq\ (Right\ Empty)\ (Stars\ [])$$

$$r_0 = (a+b)^* \xrightarrow{\backslash c_0 = b} r_1 = (\mathbf{0}+\mathbf{1})\cdot(a+b)^*$$
$$\downarrow mkeps$$
$$v_0 = Stars[Right\ b] \xleftarrow{inj_{r_0}\, c_0} v_1 = Seq\ (Right\ Empty)\ (Stars\ [])$$

Again, values such as aa are not in their fully-expanded form $(Seq\ (Char\ a)\ (Char\ a))$ for space reasons.

3.3.2 Examples on How Injection and Lexer Works

We will provide a few examples on how *inj* and *lexer* works by showing their values in each recursive call on some concrete examples. We start with the call $lexer\ (a + aa)^* \cdot c\ aac$ on the lexer, note that the value's character constructor $Char\ c$ is abbreviated as c for readability. Similarly the last derivative's sub-expression is abbreviated as $r_{=0}$[4] whose language interpretation is equivalent to that of $\mathbf{0}$ and therefore not crucial to be displayed fully expanded, as they will not be injected into.

$$\begin{array}{lll}
(a + aa)^* \cdot c & \xrightarrow{\backslash a} & ((\mathbf{1} + \mathbf{1}a)\cdot(a+aa)^*)\cdot c + \mathbf{0} \\
& \xrightarrow{\backslash a} & (((\mathbf{0} + (\mathbf{0}a + \mathbf{1}))\cdot(a+aa)^* + (\mathbf{1}+\mathbf{1}a)\cdot(a+aa)^*)\cdot \mathbf{c} + \mathbf{0}) + \mathbf{0} \\
& \xrightarrow{\backslash c} & ((r_{=0}\cdot c + \mathbf{1}) + \mathbf{0}) + \mathbf{0} \\
& \xrightarrow{mkeps} & Left(Left\ (Right\ Empty)) \\
& \xrightarrow{inj\ c} & Left\ (Left\ (Seq\ (Left\ (Seq\ (Right\ (Right\ Empty))\ Stars\ []))\ c)) \\
& \xrightarrow{inj\ a} & Left\ (Seq\ (Seq\ (Right\ (Seq\ Empty\ \mathbf{a}))\ Stars\ []\ c) \\
& \xrightarrow{inj\ a} & Seq\ (Stars\ [Right\ (Seq\ \mathbf{a}\ a)])\ c
\end{array}$$

We have assigned different colours for each character, as well as their corresponding locations in values and regular expressions. The most recently injected character is marked with a bold font. To show the details of how *inj* works, we zoom in to the second injection above, illustrating the recursive calls involved. We have marked the local part of the expression and value that was "injected into" with blue colour so it is easier to see what happened.

[4] which is equal to $((\mathbf{0} + (\mathbf{0}a + \mathbf{0}))\cdot(a+aa)^* + (\mathbf{0}+\mathbf{0}a)\cdot(a+aa)^*) + ((\mathbf{0}+\mathbf{0}a)\cdot(a+aa)^* + (\mathbf{0}+\mathbf{0}a)\cdot(a+aa)^*)$,.

$$
\begin{array}{l}
inj \quad ((\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^*) \cdot c + \mathbf{0} \quad a \\
\quad Left\ (Left\ (Seq\ (Left\ (Seq\ (Right\ (Right\ Empty))\ Stars\ []))\ c)) \\
= \\
Left\ (inj\ ((\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^*) \cdot c \quad a \quad (Left\ (Seq\ (Left\ (Seq\ (Right\ (Right\ Empty))\ Stars\ []))\ c))\ ) \\
= \\
Left\ (Seq\ (inj\ (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \quad a \quad (Left\ (Seq\ (Right\ (Right\ Empty)))))\ c) \\
= \\
Left\ (Seq\ (Seq\ (inj \quad (\mathbf{1} + \mathbf{1}a) \quad a \quad Right\ (Right\ Empty))\ Stars\ []))\ c) \\
= \\
Left\ (Seq\ (Seq\ (Right\ (inj\ \mathbf{1}a \quad a \quad (Right\ Empty))))\ Stars\ []) \\
= \\
Left\ (Seq\ (Seq\ (Right\ (Seq\ (mkeps\ \mathbf{1})\ (inj\ a\ a\ Empty))))\ Stars\ []) \\
= \\
Left\ (Seq\ (Seq\ (Right\ (Seq\ Empty\ a)))\ Stars\ [])
\end{array}
$$

We will now introduce the properties related to *inj* and *lexer*. The proofs to them have originally been found by Ausaf et al. in their 2016 work [14]. Some properties have also appeared in Ausaf's thesis [13] as lemmas and theorems. We introduce these properties, but leave out the proofs.
The central property of the *lexer* is that it gives the correct result according to POSIX rules. We present the theorem as formalised in Ausaf et al.'s AFP entry [15]:

Theorem 1.

- $s \in L\ r$ *if and only if there exists some value* v*, such that* $lexer\ r\ s = Some\ v$ *and* $(s, r) \to v$.
- *The lexer based on derivatives and injections is correct:*

$$
\begin{array}{lcl}
lexer\ r\ s = Some\ v & \iff & (r,\ s) \to v \\
lexer\ r\ s = None & \iff & \neg(\exists v.(r, s) \to v)
\end{array}
$$

Proof. We prove the first part by an induction on s. For $s = []$, $lexer\ r\ s = Some\ v$ means that r is nullable, and the *if* branch is taken in the first clause of *lexer*. This means v is equal to $mkeps\ r$. Therefore $|v| = [] \in L\ r$. For the other direction, $[] \in L\ r$ means that r is nullable, and therefore $lexer\ r\ [] = Some\ (mkeps\ r)$. Let $v = mkeps\ r$ and the RHS holds. For the inductive case we assume for any r, s and v $s \in L\ r \iff lexer\ r\ s = Some\ v \wedge (r, s) \to v$ holds. For the $\Longrightarrow$ direction, $c :: s \in L\ r$ means $s \in L\ r\backslash c$. Therefore we have v'' such that $lexer\ (r\backslash c)\ s = Some\ v''$ and $(r\backslash c, s) \to v''$. Now $lexer\ r\ (c :: s)$ is equal to $Some\ v'$, and $v' = (inj\ r\ c\ v'')$ by the definition of *lexer*. By property 2, $(r, c :: s) \to v'$. This concludes the proof. The second part is a corollary of the first proposition.

□

As we did earlier in this chapter with the matcher, one can introduce simplification on the regular expression in each derivative step. However, due to lexing, one needs to do a backward phase (w.r.t the forward derivative phase) and ensure that the values align with the regular expression at each step. Therefore one has to be careful not to break the correctness, as the injection function heavily relies on the structure of the regular expressions and values being aligned. This can be achieved by recording some extra rectification functions during the derivatives step and applying these rectifications in each run during the injection phase. With extra care one can show that POSIXness will not be affected by the simplifications listed here [14].

$$
\begin{array}{lcl}
simp\ r_1 \cdot r_2 & \stackrel{\text{def}}{=} & (simp\ r_1, simp\ r_2)\ match \\
 & & \quad case\ (\mathbf{0}, \_) \Rightarrow \mathbf{0} \\
 & & \quad case\ (\_, \mathbf{0}) \Rightarrow \mathbf{0} \\
 & & \quad case\ (\mathbf{1}, r_2') \Rightarrow r_2' \\
 & & \quad case\ (r_1', \mathbf{1}) \Rightarrow r_1' \\
 & & \quad case\ (r_1', r_2') \Rightarrow r_1' \cdot r_2' \\
simp\ r_1 + r_2 & \stackrel{\text{def}}{=} & (simp\ r_1, simp\ r_2) match \\
 & & \quad case\ (\mathbf{0}, r_2') \Rightarrow r_2' \\
 & & \quad case\ (r_1', \mathbf{0}) \Rightarrow r_1' \\
 & & \quad case\ (r_1', r_2') \Rightarrow r_1' + r_2' \\
simp\ r & \stackrel{\text{def}}{=} & r \qquad (otherwise)
\end{array}
$$

However, one can still end up with exploding derivatives, even with the simple-minded simplification rules allowed in an injection-based lexer.

3.4 A Case Requiring More Aggressive Simplifications

In this section we present a case of "evil" regular expressions that trigger exponential behaviours even in the presence of simplifications introduced in the previous section. This naturally suggests more aggressive simplifications such as removing duplicates at different levels and flattening nested alternatives. To make sure the lexing information can still be correctly retrieved, bitcodes can be used to represent the lexical information. This idea have been explored in Sulzmann and Lu's 2014 paper [77], but they implemented the simplification incorrectly such that their procedure cannot achieve the simplifications desired. We provide more intuition for why this is doable and how bitcodes can help.

If we start with the regular expression $(a^* \cdot a^*)^*$ and building just over a dozen successive derivatives w.r.t. the character a, one obtains a derivative regular expression with millions of nodes (when viewed as a tree) even with the mentioned simplifications.

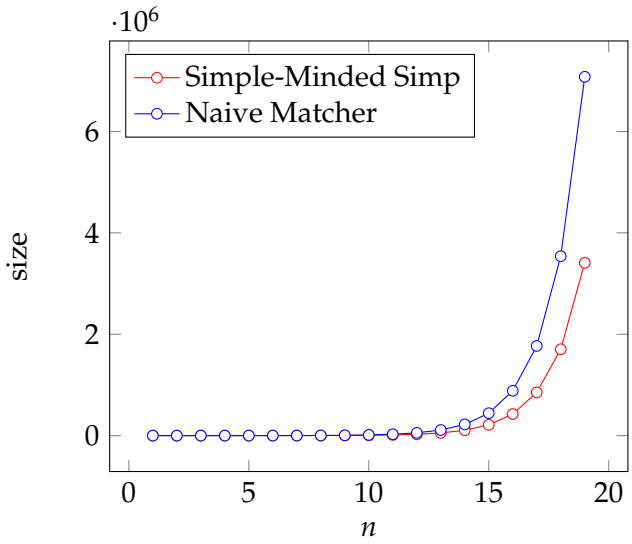


FIGURE 3.4: Size of $(a^* \cdot a^*)^*$ against $\underbrace{aa\ldots a}_{\text{n as}}$

That is because Sulzmann and Lu's injection-based lexing algorithm keeps a lot of "useless" values that will not be used. These different ways of matching will grow exponentially with the string length. Consider the case

$$r = (a^* \cdot a^*)^* \quad and \quad s = \underbrace{aa\ldots a}_{\text{n as}}$$

as an example. This is a highly ambiguous regular expression, with many ways to split up the string into multiple segments for different star iterations, and for each segment multiple ways of splitting between the two a^* sub-expressions. When n is equal to 1, there are two lexical values for the match:

$$Stars\ [Seq\ (Stars\ [Char\ a])\ (Stars\ [])] \quad (v1)$$

and

$$Stars\ [Seq\ (Stars\ [])\ (Stars\ [Char\ a])] \quad (v2)$$

The derivative of *ders_simp s r* is

$$(a^*a^* + a^*) \cdot (a^*a^*)^*.$$

The a^*a^* and a^* in the first child of the above sequence correspond to value 1 and value 2, respectively. When $n = 2$, the number goes up to 7:

$$Stars\ [Seq\ (Stars\ [Char\ a, Char\ a])\ (Stars\ [])]$$

$$Stars\ [Seq\ (Stars\ [Char\ a])\ (Stars\ [Char\ a])]$$

$$Stars\ [Seq\ (Stars\ [])\ (Stars\ [Char\ a, Char\ a])]$$

$$Stars\ [Seq\ (Stars\ [Char\ a])\ (Stars\ []), Seq\ (Stars\ [Char\ a])\ (Stars\ [])]$$

$$Stars\ [Seq\ (Stars\ [Char\ a])\ (Stars\ []), Seq\ (Stars\ [])\ (Stars\ [Char\ a])]$$

$$Stars\ [Seq\ (Stars\ [])\ (Stars\ [Char\ a]), Seq\ (Stars\ [])\ (Stars\ [Char\ a])]$$

and

$$Stars\ [Seq\ (Stars\ [])\ (Stars\ [Char\ a]), Seq\ (Stars\ [Char\ a])\ (Stars\ [])]$$

And *ders_simp aa* $(a^*a^*)^*$ is

$$((a^*a^* + a^*) + a^*) \cdot (a^*a^*)^* + (a^*a^* + a^*) \cdot (a^*a^*)^*.$$

which removes two out of the seven terms corresponding to the seven distinct lexical values.

Even with the simplifications in *ders_simp* the derivatives still grow exponentially quickly. A lexer without a good enough strategy to deduplicate will naturally have an exponential runtime on highly ambiguous regular expressions because there are exponentially many matches. For this particular example, it seems that the number of distinct matches growth speed is proportional to $(2n)!/(n!(n+1)!)$ (n being the input length).

On the other hand, the *POSIX* value for $r = (a^* \cdot a^*)^*$ and $s = \underbrace{aa\ldots a}_{\text{n } as}$ is

$$Stars\ [Seq\ (Stars\ [\underbrace{Char(a), \ldots, Char(a)}_{\text{n iterations}}]), Stars\ []].$$

At any moment, the subterms in a regular expression that will potentially result in a POSIX value is only a minority among the many other terms, and one can remove the ones that are not possible to be POSIX. In the above example,

$$((a^*a^* + \underbrace{a^*}_{\text{A}}) + \underbrace{a^*}_{\text{duplicate of A}}) \cdot (a^*a^*)^* + \underbrace{(a^*a^* + a^*) \cdot (a^*a^*)^*}_{\text{further simp removes this}}. \tag{3.1}$$

can be further simplified by removing the underlined term first, which would open up possibilities of further simplification that removes the underbraced part. The result would be

$$(\underbrace{a^*a^*}_{\text{term 1}} + \underbrace{a^*}_{\text{term 2}}) \cdot (a^*a^*)^*.$$

with corresponding values

$$\begin{array}{ll} Stars\ [Seq\ (Stars\ [Char\ a, Char\ a])\ (Stars\ [])] & (\text{term 1}) \\ Stars\ [Seq\ (Stars\ [Char\ a])\ (Stars\ [Char\ a])] & (\text{term 2}) \end{array}$$

Other terms with an underlying value, such as

$$Stars\ [Seq\ (Stars\ [])\ (Stars\ [Char\ a, Char\ a])]$$

do not to contribute a POSIX lexical value, and therefore can be thrown away.

Ausaf et al. [14] have come up with some simplification steps, and incorporated the simplification into *lexer*. They call this lexer $lexer_{simp}$ and proved that

$$lexer\ r\ s = lexer_{simp}\ r\ s$$

The function $lexer_{simp}$ involves some fiddly manipulation of value rectification, which we omit here. however those steps are not yet sufficiently strong, to achieve the above effects. And even with these relatively mild simplifications, the proof is already quite a bit more complicated than the theorem 1. One would need to prove something like this:

$$If\ (snd\ (simp\ r\backslash c), s) \rightarrow v\ \ then\ \ (r, c :: s) \rightarrow inj\ \ r\ \ c\ \ ((fst\ (simp\ r\backslash c))\ v).$$

instead of the simple lemma 2, where now *simp* not only has to return a simplified regular expression, but also what specific simplifications have been done as a function on values showing how one can transform the value underlying the simplified regular expression to the unsimplified one.

We therefore choose a slightly different approach also described by Sulzmann and Lu to get better simplifications, which uses some augmented data structures compared to plain regular expressions. We call them *annotated* regular expressions. With annotated regular expressions, we can avoid creating the intermediate values $v_1, \ldots v_n$ and a second phase altogether. We introduce this new datatype and the corresponding algorithm in the next chapter.

Chapter 4

Bit-coded Algorithm of Sulzmann and Lu

In this chapter, we are going to describe the bit-coded lexing algorithm called *blexer*, introduced by Sulzmann and Lu [77] and their correctness proof. Just like in chapter 3, the algorithms and proofs have been included for self-containedness reasons, even though they have been originally found and described by Sulzmann and Lu ([77]) and Ausaf et al. in 2016 ([14] and [13]). The earliest work on using bitcodes for lexing we are aware of dates back to 2011 from Nielsen and Henglein [54]. There is another formalisation of bitcoded regular expressions and lexing by Ribeiro and Du Bois using Agda [71]. The work looked at variants of the problem such as submatch extraction and prefix and suffix finders. The *blexer*'s proof sketches in this chapter also follows more closely the actual Isabelle formalisation. For example, lemma 3 and 6 are not included in the publications by Ausaf et al., despite them being some of the key properties leading to the correctness result. Our hope is that the reader will be able to construct an Isabelle proof on their own by just reading this chapter.

We will first motivate the bit-coded algorithm in section 4.1, and then introduce their formal definitions in section 4.2, followed by a description of the correctness proof of *blexer* in section 4.3. We call *blexer*'s induction strategy *reverse induction*, by which we mean setting the inductive case to be $s@[c]$ instead of $c :: s$.

4.1 The Motivation Behind Using Bitcodes

Let us give again the definition of *lexer* from Chapter 3:

$$\begin{array}{lcl} \mathit{lexer}\ r\ [] & = & \mathit{if}\ (\mathit{nullable}\ r)\ \mathit{then}\ \mathit{Some}(\mathit{mkeps}\ r)\ \mathit{else}\ \mathit{None} \\ \mathit{lexer}\ r\ c :: s & = & \mathit{case}\ (\mathit{lexer}\ (r\backslash c)\ s)\ \mathit{of} \\ & & \quad \mathit{None} \implies \mathit{None} \\ & & \quad \mid \mathit{Some}(v) \implies \mathit{Some}(\mathit{inj}\ r\ c\ v) \end{array}$$

This algorithm works nicely as a functional program that utilizes Brzozowski derivatives: each derivative character is remembered and stacked up, and injected back in reverse order as they have been taken derivative of. The derivative operation $\backslash$ and its reverse operation *inj* is of similar shape and compexity, and work in lockstep with each other. However if we take a closer look into the example run of *lexer* we have shown in chapter 3, many inefficiencies exist:

$$
\begin{array}{lll}
(a + aa)^* \cdot c & \stackrel{\backslash a}{\rightarrow} & ((\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^*) \cdot c + \mathbf{0} \\
& \stackrel{\backslash a}{\rightarrow} & (((\mathbf{0} + (\mathbf{0}a + \mathbf{1})) \cdot (a + aa)^* + (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^*) \cdot \mathbf{c} + \mathbf{0}) + \mathbf{0} \\
& \stackrel{\backslash c}{\rightarrow} & ((r_{=0} \cdot c + \mathbf{1}) + \mathbf{0}) + \mathbf{0} \\
& \stackrel{mkeps}{\rightarrow} & Left(Left\ (Right\ Empty)) \\
& \stackrel{inj\ c}{\rightarrow} & Left\ (Left\ (Seq\ (Left\ (Seq\ (Right\ (Right\ Empty))\ Stars\ []))\ c)) \\
& \stackrel{inj\ a}{\rightarrow} & Left\ (Seq\ (Seq\ (Right\ (Seq\ Empty\ \mathbf{a}))\ Stars\ [])\ c) \\
& \stackrel{inj\ a}{\rightarrow} & Seq\ (Stars\ [Right\ (Seq\ \mathbf{a}\ a)])\ c
\end{array}
$$

For the un

$$
\begin{array}{l}
inj \\
\quad Left\ (Left\ (Seq\ (Left\ (Seq\ (Right\ (Right\ Empty))\ Stars\ []))\ c)) \\
= \\
Left\ (inj\ ((\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^*) \cdot c \quad a \quad (Left\ (Seq\ (Left\ (Seq\ (Right\ (Right\ Empty))\ Stars\ []))\ c)) \quad) \\
= \\
Left\ (Seq\ (inj\ (\mathbf{1} + \mathbf{1}a) \cdot (a + aa)^* \quad a \quad (Left\ (Seq\ (Right\ (Right\ Empty)))))\ c) \\
= \\
Left\ (Seq\ (Seq\ (inj \quad (\mathbf{1} + \mathbf{1}a) \quad a \quad Right\ (Right\ Empty))\ Stars\ [])\ c) \\
= \\
Left\ (Seq\ (Seq\ (Right\ (inj\ \mathbf{1}a \quad a \quad (Right\ Empty))))\ Stars\ []) \\
= \\
Left\ (Seq\ (Seq\ (Right\ (Seq\ (mkeps\ \mathbf{1})\ (inj\ a\ a\ Empty))))\ Stars\ []) \\
= \\
Left\ (Seq\ (Seq\ (Right\ (Seq\ Empty\ a)))\ Stars\ [])
\end{array}
$$

After all derivatives have been taken, the stack grows to a maximum size and the pair of regular expressions and characters r_i, c_{i+1} are then popped out and used in the injection phase.

Storing all intermediate information on a stack allows the algorithm to work in an elegant way, at the expense of verbosity. The stack seems to grow at least quadratically with respect to the input (not taking into account the size bloat of r_i), which can be inefficient and prone to stack overflows.

4.2 Bitcoded Algorithm

To address this, Sulzmann and Lu defined a new datatype called *annotated regular expression*, which condenses all the partial lexing information into bitcodes. Each derivative step is accompanied by an incremental change to the bitcodes reflect a character has been matched. It becomes unnecessary to remember all the intermediate expressions, but only the most recent one with this bit-carrying regular expression. Bits and bitcodes (lists of bits) are defined as:

$$bit ::= S \mid Z \qquad bits :: "bit\ list"$$

We use S and Z rather than 1 and 0 is to avoid confusion with the regular expressions $\mathbf{0}$ and $\mathbf{1}$. One could use any single-character symbol for this, and S and Z are employed simply because they were originally used by Sulzmann and Lu in their paper when they first came up with the algorithm.

Annotated regular expressions are defined as the following datatype constructors:[1]

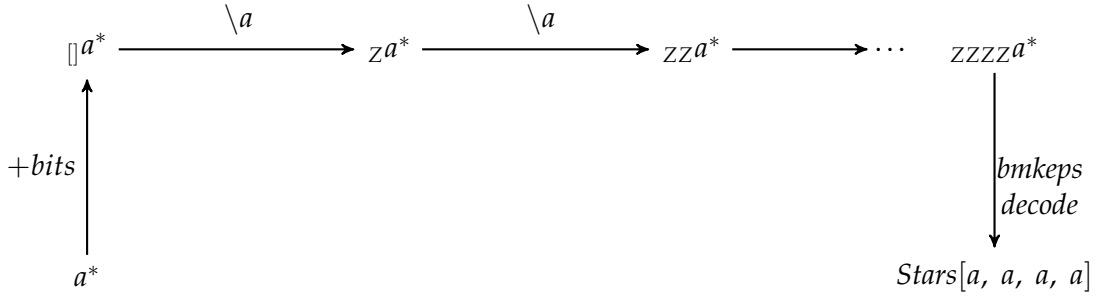
$$
\begin{array}{lcl}
a & ::= & \mathbf{0} \\
 & | & {}_{bs}\mathbf{1} \\
 & | & {}_{bs}\mathbf{c} \\
 & | & {}_{bs}\sum as \\
 & | & {}_{bs}a_1 \cdot a_2 \\
 & | & {}_{bs}a^*
\end{array}
$$

where subscripts bs stands for bit-codes, a for **a**nnotated regular expressions and as for lists of annotated regular expressions. For example the annotated version of a regular expression $a + b$ would be ${}_{[]}(_Z a +_S b)$, and the Z stands for "left", and S stands for "right". These bits are just annotations, that's why they are not written with the same font as the actual subexpressions a and b. as stands for a list of annotated regular expressions. So an alternative regular expression that was $(aba + (ab + a))$ will now represented like ${}_{[]}\sum[aba, ab, a]$. Flattening nested alternatives can be useful as that makes de-duplication more straightforward, and the nested alternative structure usually does not add any information.

The idea is to use the bitcodes to remember what happened during each derivative step. The bitcodes are thrown away if a mismatch happens, and stored and collected if partial matches are completed. For example, if we start with the annotated regular expression $(_Z a +_S b)$, and after a derivative we get ${}_Z\mathbf{1} + \mathbf{0}$, then the Z bit can be used to infer that the derivative was with respect to a, as the right alternative's bitcode is thrown away.

4.2.1 A Bird's Eye View of the Bit-coded Lexer

We start with a teaser run of a sample computation of lexing using bitcodes and derivatives, before we give the precise details of the functions and definitions related to Sulzmann and Lu's *blexer* (*b*-itcoded lexer). Suppose we want to lex aa against the regular expression a^*. The picture shows how normal regular expressions are annotated with bitcodes, and how these bitcodes change. Finally it illustrates how these bitcodes are decoded into lexical values.



The plain regular expressions are first "lifted" to an annotated regular expression by attaching bits to it. Then the annotated regular expression ${}_{[]}a^*$ will go through successive derivatives with respect to the input characters, in this example consecutive as. Each derivative adds a Z bit, which marks another iteration of the Kleene star.

[1] We use subscript notation to indicate that the bitcodes are auxiliary information that does not interfere with the structure of the regular expressions

Finally, the bitcodes are collected and decoded by *bmkeps* and *decode*, which results in the value *Stars* $[a,\ a,\ a,\ a]$ that inhabits a^*.

The most notable improvements of *blexer* over *lexer* are

- the absence of the second injection phase.
- intermediate derivatives no longer stored on a stack as *lexer* does. This saves space.
- One can optimise *blexer* without breaking lexing.

Despite *blexer*'s advantages, we need to reuse *lexer*'s correctness proof as a foundation for *blexer* (and *blexer_simp*)'s correctness. That is why *lexer* was still introduced in chapter 3. In the next section we introduce in detail all the functions used in *blexer*. All these functions can be found originally in Sulzmann and Lu's paper [77], (we do not repeat this fact when presenting each function) and we introduce them for self-containedness reasons.

4.2.2 Operations in *Blexer*

The first operation we define is how we prepend bitcodes to the front of existing bitcodes of an annotated regular expression. This operation is called *fuse*:

$$
\begin{array}{lcl}
\mathit{fuse} & :: & \mathit{bits} \to \mathit{arexp} \to \mathit{arexp} \\
\mathit{fuse}\ bs\ (\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
\mathit{fuse}\ bs\ ({}_{bs'})\mathbf{1} & \stackrel{\text{def}}{=} & {}_{bs@bs'}\mathbf{1} \\
\mathit{fuse}\ bs\ ({}_{bs'}\mathbf{c}) & \stackrel{\text{def}}{=} & {}_{bs@bs'}\mathbf{c} \\
\mathit{fuse}\ bs\ ({}_{bs'}\sum as) & \stackrel{\text{def}}{=} & {}_{bs@bs'}\sum as \\
\mathit{fuse}\ bs\ ({}_{bs'}a_1 \cdot a_2) & \stackrel{\text{def}}{=} & {}_{bs@bs'}a_1 \cdot a_2 \\
\mathit{fuse}\ bs\ ({}_{bs'}a^*) & \stackrel{\text{def}}{=} & {}_{bs@bs'}a^*
\end{array}
$$

For example, $\mathit{fuse}\ \ [Z, Z, S]\ \ {}_Z a^*$ returns ${}_{ZZSZ}a^*$. With *fuse* we are able to define the *internalise* function, written $(\_)^\uparrow$, that annotates a plain regular expression to a bitcoded regular expression. This function will be applied before we start taking derivatives. Intuitively one simply looks for all alternative regular expressions, and turn that into a 2-element list, marking the first element with Z, and the second element with S. The rest of the constructors are just marked with an empty list.

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

Here are some reasons for why such annotations are useful. For example with $r_1 + r_2$ one annotates the children with Z and S: ${}_Z r_1 +_S r_2$ so that one can use the bit collected to determine whether r_1 or r_2 was matched. If the bit corresponding to the sub-expression $r_1 + r_2$ is Z, then we know r_1 matched, and vice versa. The annotated regular expressions would look overwhelming if we explicitly indicate all the locations where bitcodes are attached. For example, $(aa)^* \cdot (b + c)$ would look like

$[](([][]a \cdot_{[]} a)^* \cdot_{[]} ([]b +_{[]} c))$ after internalise. Therefore for readability we omit bitcodes if they are empty. This applies to all annotated regular expressions in this thesis.

The opposite of *internalise* is *erase*, where all bit-codes are removed, and the alternative $\sum as$'s list of expressions as is converted back to nested binary sums.

$$\begin{array}{lcl}
\mathbf{0}_\downarrow & \stackrel{\text{def}}{=} & \mathbf{0} \\
(_{bs}\mathbf{1})_\downarrow & \stackrel{\text{def}}{=} & \mathbf{1} \\
(_{bs}\mathbf{c})_\downarrow & \stackrel{\text{def}}{=} & \mathbf{c} \\
(_{bs}a_1 \cdot a_2)_\downarrow & \stackrel{\text{def}}{=} & (a_1)_\downarrow \cdot (a_2)_\downarrow \\
(_{bs}[])_\downarrow & \stackrel{\text{def}}{=} & \mathbf{0} \\
(_{bs}[a])_\downarrow & \stackrel{\text{def}}{=} & a_\downarrow \\
(_{bs}\sum[a_1,\ a_2])_\downarrow & \stackrel{\text{def}}{=} & (a_1)_\downarrow + (a_2)_\downarrow \\
(_{bs}\sum(a :: as))_\downarrow & \stackrel{\text{def}}{=} & a_\downarrow + \ (_{[]}\sum as)_\downarrow \\
(_{bs}a^*)_\downarrow & \stackrel{\text{def}}{=} & (a_\downarrow)^*
\end{array}$$

Where we abbreviate the *erase a* operation as $(a)_\downarrow$, for conciseness. Internalise and erase are inverses of each other, and one can find a proof for that $(r^{\uparrow}_{\downarrow})$ in the AFP entry by Ausaf et al. [15].

The functions *bnullable* and *bmkeps* are routine extensions to *nullable* and *mkeps*. The only complication is that instead of returning a value, *bmkeps* returns a bitcode that encodes the value.

$$\begin{array}{lcl}
bnullable & :: & "arexp \Rightarrow bool" \\
bnullable\ a & \stackrel{\text{def}}{=} & nullable\ (a_\downarrow)
\end{array}$$

$$\begin{array}{lcl}
bmkeps & :: & "arexp \Rightarrow bits" \\
bmkeps\ (_{bs}\mathbf{1}) & \stackrel{\text{def}}{=} & bs \\
bmkeps\ (_{bs}\sum a :: as) & \stackrel{\text{def}}{=} & if\ bnullable\ a \\
 & & then\ bs \mathbin{@} bmkeps\ a \\
 & & else\ bs \mathbin{@} bmkeps\ (_{[]}\sum as) \\
bmkeps\ (_{bs}a_1 \cdot a_2) & \stackrel{\text{def}}{=} & bs \mathbin{@} bmkeps\ a_1 \mathbin{@} bmkeps\ a_2 \\
bmkeps\ (_{bs}a^*) & \stackrel{\text{def}}{=} & bs \mathbin{@} [S]
\end{array}$$

The only time when *bmkeps* creates new bitcodes is when it completes a star's iterations by attaching a S to the end of the bitcode list it returns.

The bitcodes extracted by *bmkeps* need to be *decode*d (with the guidance of a plain regular expression):

$$
\begin{array}{lll}
\mathit{decode}'\ \mathit{bs}\ (\mathbf{1}) & \stackrel{\text{def}}{=} & (\mathit{Empty}, \mathit{bs}) \\
\mathit{decode}'\ \mathit{bs}\ (c) & \stackrel{\text{def}}{=} & (\mathit{Char}\ c, \mathit{bs}) \\
\mathit{decode}'\ (Z :: \mathit{bs})\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{let}\ (v, \mathit{bs}_1) = \mathit{decode}'\ \mathit{bs}\ r_1\ \mathit{in}\ (\mathit{Left}\ v, \mathit{bs}_1) \\
\mathit{decode}'\ (S :: \mathit{bs})\ (r_1 + r_2) & \stackrel{\text{def}}{=} & \mathit{let}\ (v, \mathit{bs}_1) = \mathit{decode}'\ \mathit{bs}\ r_2\ \mathit{in}\ (\mathit{Right}\ v, \mathit{bs}_1) \\
\mathit{decode}'\ \mathit{bs}\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \mathit{let}\ (v_1, \mathit{bs}_1) = \mathit{decode}'\ \mathit{bs}\ r_1\ \mathit{in} \\
 & & \quad \mathit{let}\ (v_2, \mathit{bs}_2) = \mathit{decode}'\ \mathit{bs}_1\ r_2 \\
 & & \qquad \mathit{in}\ (\mathit{Seq}\ v_1\ v_2, \mathit{bs}_2) \\
\mathit{decode}'\ (S :: \mathit{bs})\ (r^*) & \stackrel{\text{def}}{=} & (\mathit{Stars}\ [], \mathit{bs}) \\
\mathit{decode}'\ (Z :: \mathit{bs})\ (r^*) & \stackrel{\text{def}}{=} & \mathit{let}\ (v, \mathit{bs}_1) = \mathit{decode}'\ \mathit{bs}\ r\ \mathit{in} \\
 & & \quad \mathit{let}\ (\mathit{Stars}\ vs, \mathit{bs}_2) = \mathit{decode}'\ \mathit{bs}_1\ r^* \\
 & & \qquad \mathit{in}\ (\mathit{Stars}\ v :: vs, \mathit{bs}_2) \\
 & & \\
\mathit{decode}\ \mathit{bs}\ r & \stackrel{\text{def}}{=} & \mathit{let}\ (v, \mathit{bs}') = \mathit{decode}'\ \mathit{bs}\ r\ \mathit{in} \\
 & & \quad \mathit{if}\ \mathit{bs}' = []\ \mathit{then}\ \mathit{Some}\ v\ \mathit{else}\ \mathit{None}
\end{array}
$$

The function *decode′* returns a pair consisting of a partially decoded value and some leftover bit-list. The wrapper function *decode* succeeds if the left-over bit-sequence returned by *decode′* is empty. The result type of *decode* is wrapped with an Option monad. To indicate a failed decode, *None* is returned. A few examples of *decode′* and *decode*:

- $\mathit{decode}'\ ZZSZZZS\ ((aba + ab) + a)^* = (\mathit{Stars}\ [\mathit{Left}\ (\mathit{Right}\ ab),\ \mathit{Left}\ (\mathit{Left}\ aba)], [])$.

  In the bitcode argument of the above call, the first three bits ZZS corresponds to the first iteration, and second three bits ZZZ are for the second, finally ended by an S bit. The suffix ZS in ZZS points to the middle subexpression (ab) in the internalised expression $({}_Z({}_Z aba +_S ab) +_S a)^*$, and the Z bit indicates an iteration. Similarly for ZZZ the first bit marks a new iteration and ZZ refers to to aba.

  $\mathit{decode}\ ZZSZZZS\ ((aba + ab) + a)^* = \mathit{Some}\ \mathit{Stars}\ [\mathit{Left}\ (\mathit{Right}\ ab),\ \mathit{Left}\ (\mathit{Left}\ aba)]$

  because it *decode′* used up all the bitcodes.

- $\mathit{decode}'\ ZZSZZZSZZ\ ((aba + ab) + a)^* = (\mathit{Stars}\ [\mathit{Left}\ (\mathit{Right}\ ab),\ \mathit{Left}\ (\mathit{Left}\ aba)], ZZ)$

  $$\mathit{decode}\ ZZSZZZSZZ\ ((aba + ab) + a)^* = \mathit{None}$$

  because the bitcodes returned by *decode′* is not empty.

- $$\mathit{decode}'\ a + b\ ZZ = (\mathit{Left}\ a,\ Z)$$

  $$\mathit{decode}\ a + b\ ZZ = \mathit{None}$$

  ZZ cannot not "fit into" the shape of $a + b$. *decode* is saying here "you have not correctly given me matching regular expression and bitcodes"

One might argue that the pattern matching in $decode'$ is not exhaustive (e.g. $decode'\ []\ (r_1 + r_2)$ is not defined) and therefore should be extended with a catch-all clause like

$$decode'\ \_\ \_ = (Empty, S),$$

which would cause *decode* to return *None*. This is not included in the original work by Sulzmann and Lu and nor did it cause issues for Ausaf et al.'s formalisation [14]. Therefore we leave it unchanged in our formalisation of *blexer_simp*.

The inverse operation of *decode* is *code*.

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

This function encodes a value into a bitcode by converting *Left* into Z, *Right* into S, and marks the start of any non-empty star iteration by S. Z marks the border where a star iteration terminates. This coding is lossy, as it throws away the information about characters, and 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*, but this will not be necessary for our correctness proof. Here are a few examples:

- $code\ Stars\ [a,\ a;\ a] = ZZZS$
- $code\ (Right\ Empty) = S$

Ausaf et al. had formally proven that given a value v and regular expression r with $\vdash v : r$, then we have the property that *decode* and *code* are are inverse operations of one another:

$$If \vdash v : r\ then\ decode\ (code\ v)\ r = Some(v)$$

We leave out the proof whose details can be found in [15]. Now we present the definition of the central part of Sulzmann and Lu's second lexing algorithm, the *bder* function (which stands for *b*itcoded-derivative):

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

For *bder c a*, we use the infix notation $a\backslash c$ for readability. The *bder* function tells us how regular expressions can be recursively traversed, where the bitcodes are augmented and carried around when a derivative is taken. We give the intuition behind some of the more involved cases in *bder*.

For example, in the *star* case, a derivative of ${}_{bs}a^*$ means that one more star iteration needs to be taken. We therefore need to unfold it into a sequence, and attach an additional bit Z to the front of $a\backslash c$ as a record to indicate one new star iteration is unfolded.

$$({}_{bs}a^*)\backslash c \stackrel{\text{def}}{=} {}_{bs}(\underbrace{\mathit{fuse}\,[Z]\; a\backslash c}_{\text{One more iteration}}) \cdot ({}_{[]}a^*))$$

This information will be recovered later by the *decode* function.
Another place where the *bder* function differs from derivatives on regular expressions is the sequence case:

$$\begin{array}{lll} ({}_{bs}\; a_1 \cdot a_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\; \mathit{bnullable}\, a_1 \\ & & \mathit{then}\; {}_{bs}\sum[({}_{[]}\,(a_1\backslash c)\cdot a_2), \\ & & \qquad (\mathit{fuse}\,(\mathit{bmkeps}\, a_1)\,(a_2\backslash c))] \\ & & \mathit{else}\; {}_{bs}\,(a_1\backslash c)\cdot a_2 \end{array}$$

The difference mainly lies in the *if* branch (when a_1 is *bnullable*): we use *bmkeps* to store the lexing information in a_1 before collapsing it (as it has been fully matched by string prior to c), and attach the collected bit-codes to the front of a_2 before throwing away a_1. In the injection-based lexer, r_1 is immediately thrown away in the *if* branch, the information r_1 stores is therefore lost:

$$\begin{array}{lll} (r_1 \cdot r_2)\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\; [] \in L(r_1) \\ & & \mathit{then}\; (r_1\backslash c)\cdot r_2 \;+\; r_2\backslash c \\ & & \mathit{else}\; \ldots \end{array}$$

The rest of the clauses of *bder* is rather similar to *der*.
Generalising the derivative operation with bitcodes to strings, we have

$$\begin{array}{lll} a\backslash_s [] & \stackrel{\text{def}}{=} & a \\ a\backslash (c :: s) & \stackrel{\text{def}}{=} & (a\backslash c)\backslash_s s \end{array}$$

As we did earlier, we omit the s subscript at $\backslash_s$ when there is no danger of confusion.

4.2.3 Putting Things Together

Putting these operations altogether, we obtain a lexer with bit-coded regular expressions as its internal data structures, which we call *blexer*:

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

This function first attaches bitcodes to a plain regular expression r, and then builds successive derivatives with respect to the input string s, and then test whether the result is (b)nullable. If yes, then extract the bitcodes from the nullable expression,

and decodes the bitcodes into a lexical value. If there does not exist a match between r and s, the lexer outputs *None* indicating a mismatch. An example *blexer* run:

$$r_Z : (aba + ab + a)^* \xrightarrow{+bits} a_0 : (_Z(\ _Z\text{aba} + {}_S\text{ab}\ ) +_S a)^*$$

$$\xrightarrow{\backslash a} a_1 :_Z (_Z(\ _Z\mathbf{1}\text{ba} + {}_S\mathbf{1}\text{b}\ ) +_S \mathbf{1}) \cdot a_0 \xrightarrow{\backslash b}$$

$$_Z(_Z(\ _Z\mathbf{1}\text{a} + {}_S\mathbf{1}\ ) + \mathbf{0}) \cdot a_0 +_0 (_0(_0\mathbf{0}ba +_1 \mathbf{0}b) + \mathbf{0}) \cdot a_0 \xrightarrow{\backslash a}$$

$$(_Z(\ _Z\mathbf{1} + \mathbf{0}\ ) + \mathbf{0}) \cdot a_0 + {}_{ZZS}(_Z(_Z(\ _Z\mathbf{1}\text{ba} +_S \mathbf{1}b) +_S \mathbf{1}) \cdot a_0) + \ldots$$

$$\downarrow \backslash b$$

$$(_Z(\mathbf{0} + \mathbf{0}) + \mathbf{0}) \cdot a_0 + {}_{ZZS}(_Z(_Z(\ _Z\mathbf{1}\text{a} +_S \mathbf{1}) + \mathbf{0}) \cdot a_0) + \ldots$$

$$\xrightarrow{\backslash a} \ldots + {}_{ZZS}(_Z(_Z(\ _Z\mathbf{1} +_S \mathbf{0}) +_S \mathbf{0})) \cdot a_0 + \ldots \xrightarrow{extract}$$

$$\text{ZZS}\ \text{ZZZZ} \xrightarrow[(aba+ab+a)^*]{decode} Stars\ [\ Seq\ a\ b\ ,\ Seq\ \text{a b a}\ ]$$

Ausaf and Urban [15] formally proved the correctness of the *blexer*, namely

$$blexer\ r\ s = lexer\ r\ s.$$

This was claimed but not formalised in Sulzmann and Lu's work. We introduce the proof later, after we give all the needed auxiliary functions and definitions. This is to show how such proofs break down once simplifications are applied.

4.2.4 An Example *blexer* Run

Before introducing the proof we shall first introduce the auxiliary definitions which help establish the inductive strategy. Intuitively, they form the scaffolding of *blexer*, linking each step with that of *lexer*'s. They help pin down the intuition that *blexer*'s bitcodes at every step are correct, and one can extract the bitcodes under the guidance of a value which tells which path to take to find those bitcodes. The "finding bitcodes guided by a value" function as devised by Sulzmann and Lu has the following definition:

$$\begin{array}{lll}
retrieve\ {}_{bs}\mathbf{1} & Empty & \stackrel{\text{def}}{=}\ bs \\
retrieve\ {}_{bs}\mathbf{c} & (Char\ c) & \stackrel{\text{def}}{=}\ bs \\
retrieve\ {}_{bs}a_1 \cdot a_2 & (Seq\ v_1\ v_2) & \stackrel{\text{def}}{=}\ bs\ @\ (retrieve\ a_1\ v_1)\ @\ (retrieve\ a_2\ v_2) \\
retrieve\ {}_{bs}\Sigma(a :: as) & (Left\ v) & \stackrel{\text{def}}{=}\ bs\ @\ (retrieve\ a\ v) \\
retrieve\ {}_{bs}\Sigma(a :: as) & (Right\ v) & \stackrel{\text{def}}{=}\ bs\ @\ (retrieve\ ({}_{[]}\Sigma as)\ v) \\
retrieve\ {}_{bs}a^* & (Stars\ (v :: vs)) & \stackrel{\text{def}}{=}\ bs\ @\ [Z]\ @\ (retrieve\ a\ v)\ @\ (retrieve\ ({}_{[]}a^*)\ (Stars\ vs)) \\
retrieve\ {}_{bs}a^* & (Stars\ []) & \stackrel{\text{def}}{=}\ bs\ @\ [S]
\end{array}$$

retrieve was described by Sulzmann and Lu, but it was never used in their pencil-and-paper proof. Ausaf and Urban [13] later discovered its usage and applied *retrieve* to *blexer*'s correctness.

The other auxiliary function called *flex* was invented by Ausaf et al. [14] to make reverse induction compatible with *lexer*:

$$\begin{array}{lcl}
flex\ r\ f\ []\ v & = & f\ v \\
flex\ r\ f\ (c :: s)\ v & = & flex\ r\ (\lambda v.\ f(inj\ r\ c\ v))\ \ s\ v
\end{array}$$

flex accumulates the characters that need to be injected back, and does the injection in a stack-like manner (the last character being chopped off in the derivatives phase is first injected). The inductive cases of *lexer*'s correctness proof were [] and $c :: s$, but *blexer*'s correctness proof requires [] and $s@[c]$ because *blexer* naturally works in a forward way. *flex* is composable w.r.t such splitting:

Property 3. $flex\ \ r\ \ f\ \ (s@[c])\ \ v = flex\ \ r\ \ f\ \ s\ \ (inj\ (r\backslash s)\ c\ v)$

flex makes the value v and function f that manipulates the value explicit parameters, so that v and f can be acted on and composed with other functions. The function *flex* can calculate what *lexer* calculates, given the input regular expression r, the identity function *id*, the input string s and the value $v_n = mkeps\ (r\backslash s)$:

Property 4. $flex\ \ r\ \ id\ \ s\ \ (mkeps\ (r\backslash s)) = lexer\ r\ s$

For both properties see [13] for a proof.

flex and *retrieve* are all the necessary preliminaries to connect *lexer* and *blexer*. We first walk through a concrete example of our *blexer* calculating the right lexical information, and present the "hidden" values in this process using *retrieve* and *flex*.

Consider the regular expression $(aa)^* \cdot (b + c)$ matching the string *aab*. We present again the bird's eye view of this particular example in each stage of the algorithm:

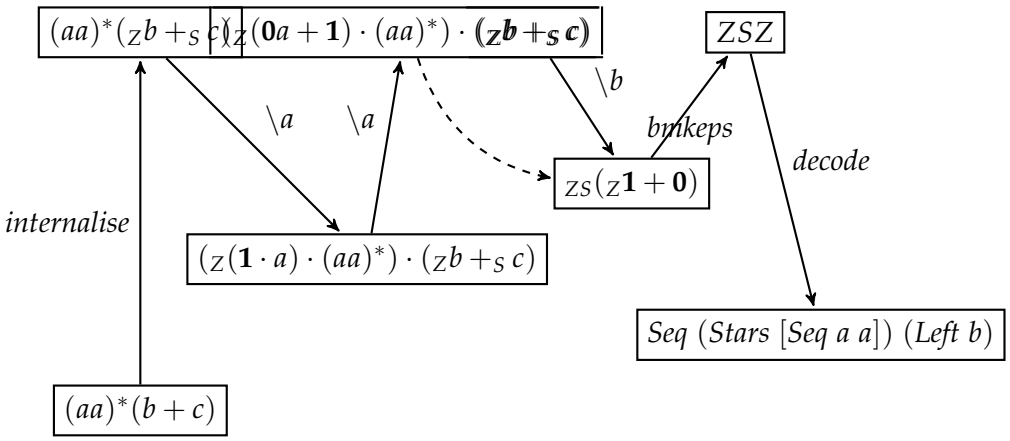


FIGURE 4.1: *blexer* with the regular expression $(aa)^*(b + c)$ and *aab*

The *internalise* function first marks the positions in an alternative with different bitcodes. For any subexpression its position-related bitcodes are unique: For instance in the above b is mapped to Z, and a to S. This uniqueness links the bitcodes with their respective values. This works for arbitrary nested alternatives, the bitcodes just get longer: for instance in the previous example $(_Z(_Z aba +_S ab) +_S a)^*$ Z refers to $aba + ab$, ZZ for aba, and S for a. *internalise*'s correctness is characterised by the following proposition

Property 5.

$$\vdash v : r \implies retrieve\ r^{\uparrow}\ v = code\ v$$

which is proven in [15]. Interestingly a slight change in the above proposition makes it untrue

$$\vdash v : (a)_{\downarrow} \implies retrieve\ a\ v = code\ v$$

because a may contain additional bits. A counterexample can be given:

$$\begin{array}{lll} \mathit{retrieve} \quad {}_{ZZZZ}(aa)^*({}_Zb +_S c) \quad (\mathit{Seq}\ (\mathit{Stars}\ [])\ (\mathit{Left}\ b)) & = \\ ZZZZSZ & \neq \\ \mathit{code} \quad (\mathit{Seq}\ (\mathit{Stars}\ [])\ (\mathit{Left}\ b)) & = \\ SZ. & \end{array}$$

In other words, *internalise* does just enough bit annotation, not too much, not too little. This way these bitcodes will uniquely determine the partial value corresponding to their own subexpressions. The examples include

- $\mathit{retrieve} \quad (aa)^*({}_Zb +_S c) \quad (\mathit{Seq}\ (\mathit{Stars}\ [])\ (\mathit{Left}\ b)) = SZ = \mathit{code}\ (\mathit{Seq}\ (\mathit{Stars}\ [])\ (\mathit{Left}\ b))$
- $\mathit{retrieve} \quad (aa)^*({}_Zb +_S c) \quad (\mathit{Seq}\ (\mathit{Stars}\ [aa])\ (\mathit{Right}\ c)) = ZSS = \mathit{code}\ (\mathit{Stars}\ [aa])\ (\mathit{Right}\ c)$
- $\mathit{retrieve} \quad ({}_Z({}_Zaba +_S ab) +_S a)^* \quad (\mathit{Left}\ (\mathit{Left}\ aba)) = ZZS = \mathit{code}\ (\mathit{Left}\ (\mathit{Left}\ aba))$

Everything during the bitcoded derivative steps are the same as *lexer*'s, except that new star iterations create new bitcodes to mark an unfold. This close alignment with *lexer* guarantees its correctness, meaning that if one take a derivative and collect the bitcodes, it will get the right bitcodes in relation to previous steps (see proof in [15]):

Property 6. $\vdash v : (a\backslash c)_\downarrow \implies \mathit{retrieve}\ a\ (\mathit{inj}\ a_\downarrow\ c\ v) = \mathit{retrieve}\ (a\backslash c)\ v$

Intuitively, this says the intermediate values in the injection phase are useful for *blexer* as "scaffolding" to confirm that the right bitcodes are created and reside in the right place. The very last retrieved bitcodes is equal to what was produced by *bmkeps*:

Property 7.

$$\mathit{bnullable}\ a \implies \mathit{bmkeps}\ a = \mathit{retrieve}\ a\ (\mathit{mkeps}\ (a_\downarrow))$$

Now these three properties 5, 6 and 7 establishes all necessary equalities to connect *blexer* and *lexer*'s results. This will become clear with the following list of all possible *retrieve*s with values from *lexer*:

$$\begin{array}{lll} ZSZ & & = \\ \mathit{code} \quad (\mathit{Seq}\ (\mathit{Stars}\ [aa])\ (\mathit{Left}\ b)) & & \overset{5}{=} \\ \mathit{retrieve} \quad (aa)^* \cdot ({}_Zb +_S c) \quad (\mathit{Seq}\ (\mathit{Stars}\ [aa])\ (\mathit{Left}\ b)) & & \overset{6}{=} \\ \mathit{retrieve} \quad ({}_Z(\mathbf{1} \cdot a) \cdot (aa)^*)({}_Zb +_S c) \quad \mathit{Seq}\ (\mathit{Seq}\ (\mathit{Seq}\ \mathit{Empty}\ a)\ (\mathit{Stars}\ [aa]))\ (\mathit{Left}\ b) & & \overset{6}{=} \\ \mathit{retrieve} \quad ({}_Z(\mathbf{0}a + \mathbf{1}) \cdot (aa)^*) \cdot ({}_Zb +_S c) \quad \mathit{Seq}\ (\mathit{Seq}\ (\mathit{Right}\ \mathit{Empty})\ (\mathit{Stars}\ []))\ (\mathit{Left}\ b) & & \overset{6}{=} \\ \mathit{retrieve} \quad {}_{ZS}(aa)^* \cdot ({}_Z\mathbf{1} + \mathbf{0}) \quad (\mathit{Seq}\ (\mathit{Stars}\ [])\ (\mathit{Left}\ \mathit{Empty})) & & \overset{7}{=} \\ \mathit{bmkeps} \quad {}_{ZS}(aa)^* \cdot ({}_Z\mathbf{1} + \mathbf{0}) & & = \\ ZSZ & & \end{array}$$

Now we have all the jigsaws in the puzzle and are ready for the correctness proof.

4.3 Correctness of the Bit-coded Algorithm (Without Simplification)

The nice thing for *blexer* is in the end there is no backward phase, as the bitcodes just requires decoding–the value information is already there. In our example, ZSZ is extracted from ${}_{ZS}({}_Z\mathbf{1} + \mathbf{0})$ from the nullable part. The lack of backward phase

also manifest itself in the correctness proof of *blexer*, where an induction case split of $s@[c]$ is used instead of $c :: s$. As we mentioned earlier while introducing *flex*, the most natural inductive case split for *lexer* is not $s@[c]$. Fortunately $lexer\ r\ s = flex\ r\ s$, therefore one simply needs to equate $flex\ r\ s$ with $blexer\ r\ s$.
flex and *blexer* calculates the same value.

Property 8. *If* $\vdash v : (r\backslash s)$, *then* $flex\ r\ id\ s\ v = decode\ (retrieve\ (r\backslash s)\ v)\ r$

We present the proof even if it can be found in [13], as it provides an alternative perspective into how *retrieve* and *flex* work (and how they break down with simplifications).

Proof. By induction on s. We prove the interesting case where both *flex* and *decode* successfully terminates with some value. We take advantage of the stepwise properties both sides. The induction tactic is reverse induction on string s. The inductive hypothesis says that $flex\ r\ id\ s\ v = decode\ (retrieve\ (r\backslash s)\ v)\ r$ holds, where v can be any value satisfying the assumption $\vdash v : (r\backslash s)$. The crucial point is to rewrite

$$retrieve\ \ (r\backslash(s@[c]))\ \ (mkeps\ (r\backslash(s@[c])))$$

as

$$retrieve\ \ (r\backslash s)\ \ (inj\ (r\backslash s)\ c\ mkeps(r\backslash(s@[c])))$$

using lemma 6. This enables us to equate

$$retrieve\ (r\backslash(s@[c]))\ (mkeps\ (r\backslash(s@[c])))$$

with

$$flex\ r\ id\ s\ (inj\ (r\backslash s)\ c\ (mkeps(r\backslash s@[c])))$$

using IH, which in turn can be rewritten as

$$flex\ r\ id\ (s@[c])\ (mkeps\ (r\backslash(s@[c])))$$

. □

With this pivotal property we can now link *flex* and *blexer* and finally give the correctness of *blexer*–it outputs the same result as *lexer*:

Theorem 2.

- $blexer\ r\ s = lexer\ r\ s$
- *The blexer function correctly implements a POSIX lexer, namely* $(r, s) \rightarrow v$ *iff* $blexer\ r\ s = Some\ v$ *and* $\nexists v.(r, s) \rightarrow v$ *iff* $blexer\ r\ s = None$

Proof. We only prove the first part as the second is a corallary from it. The interesting case is when $s \in L\ r$ and therefore *bnullable* $r^{\uparrow}\backslash s$. We have

$$blexer\ r\ s = decode\ (retrieve\ (\ r^{\uparrow})\ (mkeps\ (r\backslash s)))\ r$$

by property 7. RHS is equal to

$$flex\ r\ id\ s\ (mkeps\ (r\backslash s)),$$

by property 8, which in turn equals

$$lexer\ r\ s.$$

from property 4. □

Our main reason for analysing the bit-coded algorithm over the injection-based one is that it allows us to define more aggressive simplifications. We will elaborate on this in the next chapter.

Chapter 5

Correctness of Bit-coded Algorithm with Simplification

This chapter is the point from which novel contributions of this PhD project starts. The material in the previous chapters is necessary for this thesis, because it provides the context for why we need a new framework for the proof of our bitcoded lexer with simplifications, called *blexer_simp*.

Sulzmann and Lu implemented an optimisation for *blexer*, which we call *blexer_SLSimp*. They believe *blexer_SLSimp* achieves "linear time complexity":

> *"Linear-Time Complexity Claim*
> *It is easy to see that each call of one of the functions/operations: simp, fuse, mkEpsBC and isPhi leads to subcalls whose number is bound by the size of the regular expression involved. We claim that thanks to aggressively applying simp this size remains finite. Hence, we can argue that the above mentioned functions/operations have constant time complexity which implies that we can incrementally compute bit-coded parse trees in linear time in the size of the input."*

but in reality even with optimisations *blexer_SLSimp* still remains exponential. We discovered this complexity bug in their algorithm while trying to formalise it, and fixed those flaws with our simplification. With our formallly verified *blexer_simp* in place, we were able to fulfill the "constant time per operation" claim made by Sulzmann and Lu, this almost gives linear time complexity. (We discuss why we believe this is not yet a full-blown linear time complexity guarantee in the next chapter.) This constant bound is also formalised. The correctness proof of *blexer* is the focus of this chapter.

We believe this is yet another case for the importance of formalising algorithms. The authors have developed the lexers with the catastrophic backtracking problem in mind, and therefore their solution was targeting an efficient procedure that keeps the compuation linear. And yet their approach still contained a complexity bug that triggered the very behaviour they have set out to avoid. It is only after a rigorous proof that we finally have confidence that part of the original claims about the algorithm hold–that each derivative operation takes constant time. It seems that the gap between an intuitive complexity conjecture to a formally proven complexity property is quite often larger than expected.

We will first introduce why aggressive simplifications are needed, after which we provide our algorithm, contrasting with Sulzmann and Lu's simplifications. We then explain how our simplifications make reusing *blexer*'s correctness proof impossible. We discuss possible fixes such as rectification functions and then introduce our proof, which uses a weaker inductive invariant than those properties in chapter 4.

5.1 Ideas behind Sulzmann and Lu's Simplifications

The algorithms *lexer* and *blexer* work beautifully as functional programs, but not as practical code. One main reason for the slowness is due to the size of intermediate representations–the derivative regular expressions tend to grow unbounded if the matching involved a large number of possible matches. The successive derivative steps *lexer* and *blexer* will traverse once each of these intermediate regular expressions generated. The traversal time will be proportional to the size, as the derivative function $\backslash$ is a recursive function that visits every node in an expression. The overall computational complexity of these derivative-based lexers will be greater than or equal to the cumulative sum of traversal time in each derivative step. Therefore, an exponential size blowup with respect to input length will indicate at least exponential runtime complexity.

Consider the derivatives of the following example $(a^*a^*)^*$:

$$\begin{array}{lll} (a^*a^*)^* & \xrightarrow{\backslash a} & (a^*a^* + a^*) \cdot (a^*a^*)^* \\ & \xrightarrow{\backslash a} & ((a^*a^* + a^*) + a^*) \cdot (a^*a^*)^* + (a^*a^* + a^*) \cdot (a^*a^*)^* \\ & \xrightarrow{\backslash a} & \ldots \end{array}$$

From the second derivative several duplicate sub-expressions already needs to be eliminated (possible bitcodes are omitted to make the presentation more concise because they are not the key part of the simplifications). A simple-minded simplification function cannot simplify the third regular expression in the above chain of derivative regular expressions, namely

$$((a^*a^* + a^*) + a^*) \cdot (a^*a^*)^* + (a^*a^* + a^*) \cdot (a^*a^*)^*$$

because the duplicates are not next to each other, and therefore the rule $r + r \rightarrow r$ from *simp* does not fire. One would expect a better simplification function to work in the following way:

$$((a^*a^* + \underbrace{a^*}_{\text{A}}) + \underbrace{a^*}_{\text{duplicate of A}}) \cdot (a^*a^*)^* + \underbrace{(a^*a^* + a^*) \cdot (a^*a^*)^*}_{\text{further simp removes this}}.$$

$$\downarrow (1)$$

$$(a^*a^* + a^* + a^*) \cdot (a^*a^*)^* + \underbrace{(a^*a^* + a^*) \cdot (a^*a^*)^*}_{\text{further simp removes this}}$$

$$\downarrow (2)$$

$$(a^*a^* + a^*) \cdot (a^*a^*)^* + (a^*a^* + a^*) \cdot (a^*a^*)^*$$

$$\downarrow (3)$$

$$(a^*a^* + a^*) \cdot (a^*a^*)^*$$

In the first step, the nested alternative regular expression $(a^*a^* + a^*) + a^*$ is flattened into $a^*a^* + a^* + a^*$. Now the third term a^* can clearly be identified as a duplicate and therefore removed in the second step. This causes the two top-level terms to become the same and the second $(a^*a^* + a^*) \cdot (a^*a^*)^*$ removed in the final step. Sulzmann and Lu's simplification was designed to target such cases and their function attemps to achieve the above steps. The definitions (using our notations) are:

$$
\begin{array}{lcl}
simp\_SL\ {}_{bs}({}_{bs'}\mathbf{1} \cdot r) & \stackrel{\text{def}}{=} & \textit{if } (\textit{zeroable } r) \textit{ then } \mathbf{0} \\
 & & \textit{else } \textit{fuse } (bs@bs')\ r \\
simp\_SL\ ({}_{bs}r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \textit{if } (\textit{zeroable } r_1 \textit{ or zeroable } r_2) \textit{ then } \mathbf{0} \\
 & & \textit{else } {}_{bs}((simp\_SL\ r_1) \cdot (simp\_SL\ r_2)) \\
simp\_SL\ {}_{bs}\sum[] & \stackrel{\text{def}}{=} & \mathbf{0} \\
simp\_SL\ {}_{bs}\sum(({}_{bs'}\sum rs_1) :: rs_2) & \stackrel{\text{def}}{=} & {}_{bs}\sum((map\ (fuse\ bs')\ rs_1)@rs_2) \\
simp\_SL\ {}_{bs}\sum[r] & \stackrel{\text{def}}{=} & fuse\ bs\ (simp\_SL\ r) \\
simp\_SL\ {}_{bs}\sum(r :: rs) & \stackrel{\text{def}}{=} & {}_{bs}\sum(distinct\ (filter\ (\neg zeroable)\ ((simp\_SL\ r) :: map\ simp\_SL\ rs)))
\end{array}
$$

The *zeroable* predicate tests whether the regular expression is equivalent to $\mathbf{0}$, and can be defined as:

$$
\begin{array}{lcl}
zeroable\ {}_{bs}\sum(r :: rs) & \stackrel{\text{def}}{=} & zeroable\ r \ \wedge \ zeroable\ {}_{[]}\sum rs \\
zeroable\ {}_{bs}(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & zeroable\ r_1 \ \vee \ zeroable\ r_2 \\
zeroable\ {}_{bs}r^* & \stackrel{\text{def}}{=} & false \\
zeroable\ {}_{bs}c & \stackrel{\text{def}}{=} & false \\
zeroable\ {}_{bs}\mathbf{1} & \stackrel{\text{def}}{=} & false \\
zeroable\ {}_{bs}\mathbf{0} & \stackrel{\text{def}}{=} & true
\end{array}
$$

They also suggested that the *simp_SL* function should be applied repeatedly until a fixpoint is reached. We call this construction *SLSimp*:

$$
\begin{array}{lcl}
SLSimp\ r & \stackrel{\text{def}}{=} & while((simp\_SL\ r) \neq r) \\
 & & \quad r := simp\_SL\ r \\
 & & return\ r
\end{array}
$$

We call the operation of alternatingly applying derivatives and simplifications (until the string is exhausted) Sulz-simp-derivative, written $\backslash_{SLSimp}$:

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

After the derivatives have been taken, the bitcodes are extracted and decoded in the same manner as *blexer*:

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

We implemented this lexing algorithm in Scala, and (surprisingly) found that the final derivative regular expression size still grows exponentially under some "evil" regular expressions (note the logarithmic scale):

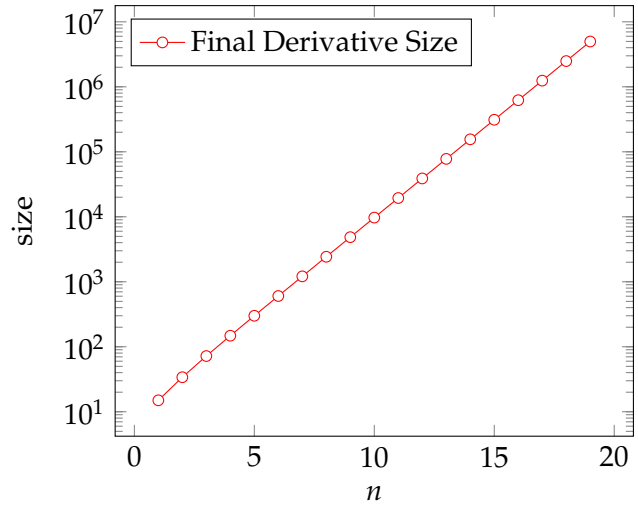


FIGURE 5.1: Lexing the regular expression $(a^*a^*)^*$ against strings of the form $\underbrace{aa\ldots a}_{\text{n as}}$ using Sulzmann and Lu's lexer

At $n = 20$ we already get an out-of-memory error with Scala's normal JVM heap size settings. In fact their simplification exhibits similar exponential behaviour as the lexer with non-structural simplifications we have shown in 3.4. The time required also grows exponentially:

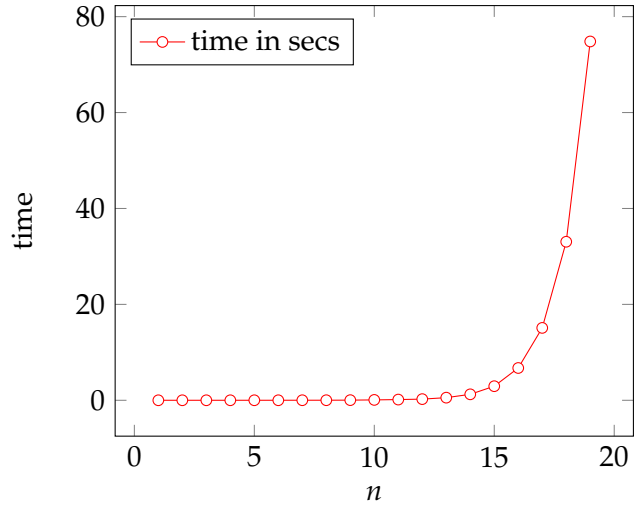


FIGURE 5.2: Lexing the regular expression $(a^*a^*)^*$ against strings of the form $\underbrace{aa\ldots a}_{\text{n as}}$ using Sulzmann and Lu's lexer

We obtain a math formula for the growth pattern of *blexer_SLSimp* in table 5.1:

$$S_n = [\![(a^*a^*)^* \backslash_{SLSimp} \underbrace{aa\ldots a}_{\text{n as}}]\!] = 2^{n-2} * 33 + 2(x \geq 3),$$

which indicates an exponential time complexity. A more detailed mathematical analysis can be found in [58], where Minamide et al. used tree transducers to calculate the asymptotic complexity of such evil regular expressions' matching time. The fact that derivatives are ever-growing falsifies Sulzmann and Lu's assumption that the size of the regular expressions in the algorithm would stay below a constant bound.

TABLE 5.1: Comparison of Derivative Sizes of $(a^*a^*)^*$ matching $\underbrace{aa\ldots a}_{\text{n as}}$

n	*blexer*	*blexer_SLSimp*
0	6	6
1	19	15
2	54	33
3	129	68
4	284	138
5	599	278
6	1234	558
7	2509	1118
8	5064	2238
9	10179	4478
10	20414	8958
11	40889	17918
12	81844	35838
13	163759	71678
14	327594	143358
15	655269	286718
16	1310624	573438
17	2621339	1146878
18	5242774	2293758
19	10485649	4587518

We will now briefly describe why *simp_SL* does not work. The problem is that the simplification here do not simplify thoroughly and consistently. For instance,

$$simp\_SL\ {}_{bs}\sum((\,{}_{bs'}\sum rs_1) :: rs_2) \quad \stackrel{\text{def}}{=} \quad {}_{bs}\sum((map\ (fuse\ bs')\ rs_1)@rs_2)$$

clause only flattens the alternative at the head position.Therefore $(a + b) + (c + d)$ will only be flattened into $a + b + (c + d)$. To flatten all elements in the list one needs something like

$$\begin{array}{lll} simp\_SL'\ {}_{bs}\sum rs & \stackrel{\text{def}}{=} & {}_{bs}\sum(flatMap \quad (\lambda r.\ r\ match\{ \\ & & \quad case\ {}_{bs'}\sum rs_1 \Rightarrow map\ (fuse\ bs')\ rs_1 \\ & & \quad case\ r \Rightarrow [r]\}) \quad rs). \end{array}$$

which first turns every regular expression into a list, and then concatenate all these lists. The second problem is that the flattened list cannot be efficiently de-duplicated by a normal *distinct* function (*nub* in their paper which is in Haskell) because they seek exact equality rather than modulo bitcodes. For example, a simple de-duplication cannot simplify ${}_{SZZ}a^* +_{SZS} a^*$ into ${}_{SZZ}a^*$ because the two sub-expressions have different annotations.

Last but not least, the *simp_SL* function is applied repeatedly in each derivative step until a fixed point is reached, which makes the algorithm even more unpredictable and inefficient. It is be much more desirable to finish the simplification process in one pass.

5.2 Our *Simp* Function

We will now introduce our own simplification function, called *bsimp*. *bsimp* does a pattern match on the input, and is called recursively on sub-expressions. The focus is on simplifying alternative regular expressions, as they are the biggest source of growth. *bsimp* first simplifies the children of an alternative, and then feeds the result expression list to a flattening function and then a de-duplication function. *bsimp* will then test the length of the list result, and remove the wrapper alternative if it is a singleton list.

bsimp will be integrated into *blexer* in the same way as *simp_SL*, namely, applied after each derivative step. We will formally prove that the after-integration lexer *blexer_simp* is correct. This means that *blexer_simp r s* $=$ *blexer r s*. But for this section we first describe the intuition why de-duplication will not cause the lexer to change its output.

Note that the function *bsimp* is by no means the optimal procedure, but it enjoys a set of nice properties such as idempotency (see lemma 10) and bounded derivative size. Further optimisations are possible and candidates include those introduced in chatper 7.

5.2.1 Flattening Nested Alternatives

The previously proposed modification to the last clause of *simp_SL* have been turned into a small flatten function so that it not only flattens the entire list, but also removes **0**s which do not contribute to the lexing result:

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

Here are some examples of *flts* computing:

$$\begin{array}{rl} & \mathit{flts}\quad {}_{ZZZ}(_S(\mathbf{0} +_Z \mathbf{1}) + a) = {}_{ZZZSZ}\ \mathbf{1} + a \\ & \mathit{flts}\quad a +_{ZZZ} ({}_{SZ}\mathbf{0} +_{SZ} \mathbf{1}) + a = a +_{ZZZSZ} \mathbf{1} + a \\ \mathit{flts} & \quad a +_{ZZZ} (_S(\mathbf{0} +_Z \mathbf{1}) + a) = a +_{ZZZS} (\mathbf{0} +_Z \mathbf{1}) +_{ZZZ} a \end{array}$$

flts only flattens same-level alternatives, as can be seen from the last example. It is not immediately obvious how this will fully simplify things. Intuitively this is because *bsimp* will recursively call *flts* to the inner regular expressions, and therefore *bsimp* will never call *flts* on the last triply nested example. See lemma 10 in chapter 6 for a formal proof that *flts* and *bsimp* is indeed sufficient to simplify things in one go.

5.2.2 Duplicate Removal

After flattening is done, an alternative is ready for de-duplication. The de-duplicate function is called *distinctBy*, and that is where we make our second improvement over Sulzmann and Lu's simplification method. The *distinctBy* function is defined as:

$$\begin{array}{lcl} \mathit{distinctBy} & :: & "'a\ \mathit{list} \Rightarrow ('a \Rightarrow' b) \Rightarrow ('b\ \mathit{set}) \Rightarrow' a\ \mathit{list} \\ \mathit{distinctBy}\ []\ f\ \mathit{acc} & = & [] \\ \mathit{distinctBy}\ (x :: \mathit{xs})\ f\ \mathit{acc} & = & \quad \mathit{if}\ (f\ x \in \mathit{acc})\ \ \mathit{then}\ \ \mathit{distinctBy}\ \mathit{xs}\ f\ \mathit{acc} \\ & & \quad \mathit{else}\ \ x :: (\mathit{distinctBy}\ \mathit{xs}\ f\ (\{f\ x\} \cup \mathit{acc})) \end{array}$$

This polymorphic function works by checking if the list head x is already in the set *acc* (accumulator). If $x \in acc$ then x is removed from the output, otherwise it is kept, and its image $f\ x$ is put into *acc*. Equivalent elements x' in this list ($f\ x' = f\ x$) which occurs later in xs will all be removed. Here is a detailed computation example:

$$\begin{array}{ll}
distinctBy\ \ [_{ZSZ}a^*a^*,\ _{ZSS}b,\ _{SZZ}a^*a^*]\ \ rerases\ \ \varnothing & = \\
_{ZSZ}a^*a^* :: (distinctBy\ \ [_{ZSS}b,\ _{SZZ}a^*a^*]\ \ rerases\ \ \{a^*a^*\}) & = \\
_{ZSZ}a^*a^* ::_{ZSS} b :: (distinctBy\ \ [_{SZZ}a^*a^*]\ \ rerases\ \ \{a^*a^*,\ b\}) & = \\
_{ZSZ}a^*a^* ::_{ZSS} b :: (distinctBy\ \ []\ \ rerases\ \ \{a^*a^*,\ b\}) & = \\
_{ZSZ}a^*a^* ::_{ZSS} b :: [] & =
\end{array}$$

In the above example, the second a^*a^* can be eliminated without affecting the language of the underlying alternative regular expression. Moreover, the overall lexing result of *blexer_simp* will still be the same. Intuitively this is because *blexer_simp* always picks up the first instance of the same subexpression. Abusing the notation slightly, after some arbitrary next derivative steps, the list will be something like $a^*a^*\backslash s + b\backslash s + a^*a^*\backslash s$, and therefore *bmkeps* will never pick up the second $a^*a^*\backslash s$ if the first is already nullable. The subexpression to the left is always the POSIX value, either because it represents a longer initial submatch, or because it is the preferred subexpression according to the left priority rule. For instance, the two terms have different inital match length (1 v.s. 0), and therefore the first expression is put to the left of the second.

$$\begin{array}{rl}
Seq\ (Stars\ [a])\ (Stars\ [a]) \text{ corresponds to } & {}_{ZSZ}\ \underbrace{a^*}_{\substack{ZS:\ match\ 1\ times}}\ \ \underbrace{a^*}_{\substack{Z:\ match\ 1\ times}} \\
Seq\ (Stars\ [])\ (Stars\ [a,\ a]) \text{ corresponds to } & {}_{SZZ}\ \underbrace{a^*}_{\substack{S:\ match\ 0\ times}}\ \ \underbrace{a^*}_{\substack{ZZ:\ match\ 2\ times}}
\end{array}$$

This idea is to be rigorously formalised and proven in the next section. The function f can be designed to involve more de-duplications, for example by just checking language equivalence rather than exact structural equality. Maybe f is defined in such a way that $[a \cdot a^* + \mathbf{1},\ a^*]$ can be turned into just $[a^*]$. A much richer set of rules can be employed, for example from Kleene algebra.

We give the definitions of *rerase* here together with the new datatype used by *rerase* (as our plain regular expression datatype does not allow non-binary alternatives). For now we can think of *rerase* as the function erase $(\_)_{\downarrow}$ defined in chapter 4 and *rrexp* as plain regular expressions, but having a general list constructor for alternatives:

$$rrexp ::= \mathbf{0}_r \mid \mathbf{1}_r \mid \mathbf{c}_r \mid r_1 \cdot r_2 \mid \sum rs \mid r^*$$

FIGURE 5.3: *rrexp*: plain regular expressions, but with $\sum$ alternative constructor

The function *rerase* we define as follows:

$$\begin{array}{lcl}
(\mathbf{0})_{\downarrow_r} & \stackrel{\text{def}}{=} & \mathbf{0}_r \\
(_{bs}\mathbf{1})_{\downarrow_r} & \stackrel{\text{def}}{=} & \mathbf{1}_r \\
(_{bs}\mathbf{c})_{\downarrow_r} & \stackrel{\text{def}}{=} & \mathbf{c}_r \\
(_{bs}r_1 \cdot r_2)_{\downarrow_r} & \stackrel{\text{def}}{=} & (r_1)_{\downarrow_r} \cdot (r_2)_{\downarrow_r} \\
(_{bs}\sum as)_{\downarrow_r} & \stackrel{\text{def}}{=} & \sum map\ (_)_{\downarrow_r}\ as \\
(_{bs}a^*)_{\downarrow_r} & \stackrel{\text{def}}{=} & (a)^*_{\downarrow_r}
\end{array}$$

We will provide more details in 6.2.1 for why a new erase function and new datatype is needed. But briefly speaking it is for backward-compatibility with *blexer*'s correctness proof and the path we (naturally) took during our proof engineering of the finiteness property.

5.2.3 Putting Things Together

We can now provide the definition of *bsimp*:

$$\begin{array}{lcl} bsimp\ ({}_{bs}a_1 \cdot a_2) & \stackrel{\text{def}}{=} & bsimp_{ASEQ}\ bs\ (bsimp\ a_1)\ (bsimp\ a_2) \\ bsimp\ ({}_{bs}\sum as) & \stackrel{\text{def}}{=} & bsimp_{ALTS}\ bs\ (distinctBy\ (flatten(map\ bsimp\ as))\ rerase\ \varnothing) \\ bsimp\ a & \stackrel{\text{def}}{=} & a \qquad otherwise \end{array}$$

The simplification (named *bsimp* for *b*it-coded) does a pattern matching on the regular expression. When it detects 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 turns into $\mathbf{0}$ or $\mathbf{1}$, which might trigger further simplification at the current level. Current level simplifications are handled by the function $bsimp_{ASEQ}$, using rules such as $\mathbf{0} \cdot r \to \mathbf{0}$ and $\mathbf{1} \cdot r \to r$.

$$\begin{array}{lcl} bsimp_{ASEQ}\ bs\ a\ b & \stackrel{\text{def}}{=} & (a,\ b) match \\ & & \quad case\ (\mathbf{0}, \_) \Rightarrow \mathbf{0} \\ & & \quad case\ (\_, \mathbf{0}) \Rightarrow \mathbf{0} \\ & & \quad case\ ({}_{bs1}\mathbf{1}, a_2') \Rightarrow fuse\ (bs @ bs_1)\ a_2' \\ & & \quad case\ (a_1', a_2') \Rightarrow {}_{bs}\ a_1' \cdot a_2' \end{array}$$

The most involved part is the $\sum$ clause, where *flts* and *distinctBy* are used together to get maximum simplification:

$$rs \xrightarrow{flts} rs_{flat} \xrightarrow{distinctBy\ rs_{flat}\ rerase\ \varnothing} rs_{distinct}$$

Here are some examples:

$$[{}_{SS}a,\ {}_{ZZZS}(\mathbf{0} +_Z \mathbf{1}),\ {}_{ZZZ}a] \xrightarrow{flts} [{}_{SS}a,\ {}_{ZZZSZ}\mathbf{1},\ {}_{ZZZ}a] \xrightarrow{distinctBy\ rs_{flat}\ rerase\ \varnothing} [{}_{SS}a,\ {}_{ZZZSZ}\mathbf{1}]$$

Finally, depending on whether the regular expression list as' has turned into a singleton or empty list after *flts* and *distinctBy*, $bsimp_{ALTS}$ decides whether to keep the current level constructor $\sum$ as it is, and removes it when there are fewer than two elements:

$$\begin{array}{lcl} bsimp_{ALTS}\ bs\ as' & \stackrel{\text{def}}{=} & as'\ match \\ & & \quad case\ [] \Rightarrow \mathbf{0} \\ & & \quad case\ a :: [] \Rightarrow fuse\ bs\ a \\ & & \quad case\ as' \Rightarrow {}_{bs}\ \sum as' \end{array}$$

Therefore one could never get for example $\sum[{}_Z a^*]$ out of *bsimp* as $bsimp_{ALTS}$ will turn it into ${}_Z a^*$. Here are a few examples of *bsimp* computation, showing some of the intermediate recursive steps:
Example 1:

$$\begin{array}{c} bsimp\ {}_{SZ}(a +_{SSS} a \cdot \mathbf{0}) \\ \downarrow \\ bsimp_{ALTS}\ [SZ]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [bsimp\ a,\ bsimp\ {}_{SSS}a \cdot \mathbf{0}])\ \varnothing) \end{array}$$

$$\downarrow$$
$$bsimp_{ALTS}\ [SZ]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [a,\ bsimp_{ASEQ}\ sss\ a\ \mathbf{0}])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [SZ]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [a,\ \mathbf{0}])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [SZ]\ (distinctBy\ (\_)_{\downarrow_r}\ [a]\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [SZ]\ [a]$$
$$\downarrow$$
$$sz a$$

Example 2:

$$bsimp\ s(z(a +_{sss}\ a \cdot \mathbf{0}) + (zz a + (zs b +_{sz}\ a)))$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [(bsimp\ z(a +_{sss}\ a \cdot \mathbf{0})),\ bsimp\ (zz a + (zs b +_{sz}\ a))])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [bsimp_{ALTS}\ [Z]\ (distinctBy\ (\_)_{\downarrow_r}\ [a]\ \varnothing),\ bsimp\ (zz a + (zs b +_{sz}\ a))])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [z a,\ bsimp\ (zz a + (zs b +_{sz}\ a))])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [z a,\ bsimp_{ALTS}\ []\ distinctBy\ (\_)_{\downarrow_r}\ (flts\ [bsimp\ zz a,\ bsimp\ (zs b +_{sz}\ a)])\ \varnothing])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [z a,\ bsimp_{ALTS}\ []\ distinctBy\ (\_)_{\downarrow_r}\ (flts\ [zz a,\ zs b,\ sz a])\ \varnothing])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [z a,\ bsimp_{ALTS}\ []\ distinctBy\ (\_)_{\downarrow_r}\ [zz a,\ zs b,\ sz a]\ \varnothing])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [z a,\ bsimp_{ALTS}\ []\ [zz a,\ zs b]])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ (flts\ [z a,\ \sum[zz a,\ zs b]])\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ (distinctBy\ (\_)_{\downarrow_r}\ [z a,\ zz a,\ zs b]\ \varnothing)$$
$$\downarrow$$
$$bsimp_{ALTS}\ [S]\ [z a,\ zs b]$$
$$\downarrow$$
$$sz a +_{szs}\ b$$

Example 3:

$$bsimp\ s(z(a \cdot (sss a \cdot \mathbf{0})) \cdot (zz a \cdot (zs b \cdot_{sz}\ a)))$$
$$\downarrow$$
$$bsimp_{ASEQ}\ [S]\ (bsimp\ z(a \cdot (sss a \cdot \mathbf{0})))\ (bsimp\ (zz a \cdot (zs b \cdot_{sz}\ a)))$$
$$\downarrow$$
$$bsimp_{ASEQ}\ [S]\ (bsimp_{ASEQ}\ [Z]\ (bsimp\ a)\ (bsimp\ sss a \cdot \mathbf{0}))\ (bsimp\ (zz a \cdot (zs b \cdot_{sz}\ a)))$$
$$\downarrow$$
$$bsimp_{ASEQ}\ [S]\ (bsimp_{ASEQ}\ [Z]\ a\ \mathbf{0})\ (bsimp\ (zz a \cdot (zs b \cdot_{sz}\ a)))$$
$$\downarrow$$
$$bsimp_{ASEQ}\ [S]\ \mathbf{0}\ (bsimp\ (zz a \cdot (zs b \cdot_{sz}\ a)))$$

$$
\begin{array}{c}
\downarrow \\
bsimp_{ASEQ}\ [S]\ \mathbf{0}\ (bsimp_{ASEQ}\ []\ (bsimp\ _{ZZ}a)\ (bsimp\ (_{ZS}b \cdot_{SZ} a))) \\
\downarrow \\
bsimp_{ASEQ}\ [S]\ \mathbf{0}\ (bsimp_{ASEQ}\ []\ _{ZZ}a\ (_{ZS}b \cdot_{SZ} a)) \\
\downarrow \\
bsimp_{ASEQ}\ [S]\ \mathbf{0}\ (_{ZZ}a \cdot (_{ZS}b \cdot_{SZ} a)) \\
\downarrow \\
\mathbf{0}
\end{array}
$$

Note that we sometimes use the $\sum$ notation and sometimes the infix $+$ notation for alternatives, whichever makes the presentation most clear and understandable. Integrating *bsimp* into *blexer* is the same as *simp_SL*, by adding it as a phase after a derivative is taken:

$$a\backslash_{bsimp} c \stackrel{\text{def}}{=} bsimp(a\backslash c)$$

similarly an extension from characters to strings is routine:

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

The lexer that extracts bitcodes from the derivatives with simplifications from our *simp* function is called *blexer_simp*:

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

Here are a few concrete examples showing the recursive computation for *blexer_simp*:
Example 1:

$$
\begin{array}{c}
(aa)^* \stackrel{(\_)^{\uparrow}}{\longrightarrow} (aa)^* \stackrel{\backslash a}{\longrightarrow}_Z (\mathbf{1}a) \cdot (aa)^* \stackrel{bsimp}{\longrightarrow} (_Z a) \cdot (aa)^* \stackrel{\backslash a}{\longrightarrow} (_Z\mathbf{1}) \cdot (aa)^* \stackrel{bsimp}{\longrightarrow}_Z (aa)^* \\
\stackrel{\backslash a}{\longrightarrow}_Z (_Z(\mathbf{1}a) \cdot (aa)^*) \stackrel{bsimp}{\longrightarrow}_Z ((_Z a) \cdot (aa)^*) \stackrel{\backslash a}{\longrightarrow}_Z ((_Z\mathbf{1}) \cdot (aa)^*) \stackrel{bsimp}{\longrightarrow}_{ZZ} (aa)^* \\
\stackrel{bmkeps}{\longrightarrow} ZZS \stackrel{decode\ (aa)^*}{\longrightarrow} Stars\ [aa,\ aa]
\end{array}
$$

Example 2:

$$
\begin{array}{c}
(a^*a^*)^* \stackrel{(\_)^{\uparrow}}{\longrightarrow} (a^*a^*)^* \stackrel{\backslash a}{\longrightarrow}_Z (((_Z\mathbf{1}) \cdot a^*) \cdot a^* +_S (_Z\mathbf{1}a^*)) \cdot (a^*a^*)^* \stackrel{bsimp}{\longrightarrow} \\
{}_Z(((_Z a^*)a^*) +_{SZ} a^*) \cdot (a^*a^*)^* \stackrel{\backslash a}{\longrightarrow}_Z \\
(((_Z(_Z\mathbf{1}a^*) \cdot a^* +_{ZS} (_Z\mathbf{1}a^*)) +_{SZ} (_Z\mathbf{1}a^*)) \cdot (a^*a^*)^*) + \\
{}_{ZZZSSS}(_Z(((_Z\mathbf{1}) \cdot a^*) \cdot a^* +_S (_Z\mathbf{1}a^*)) \cdot (a^*a^*)^* \stackrel{bsimp}{\longrightarrow} \\
(_{ZZ}a^*a^* +_{ZSZ} a^*) \cdot (a^*a^*)^* \stackrel{bmkeps}{\longrightarrow} ZZSSS \stackrel{decode\ (a^*a^*)^*}{\longrightarrow} Stars\ [Seq\ (Stars\ [a,\ a]),\ (Stars\ [])]
\end{array}
$$

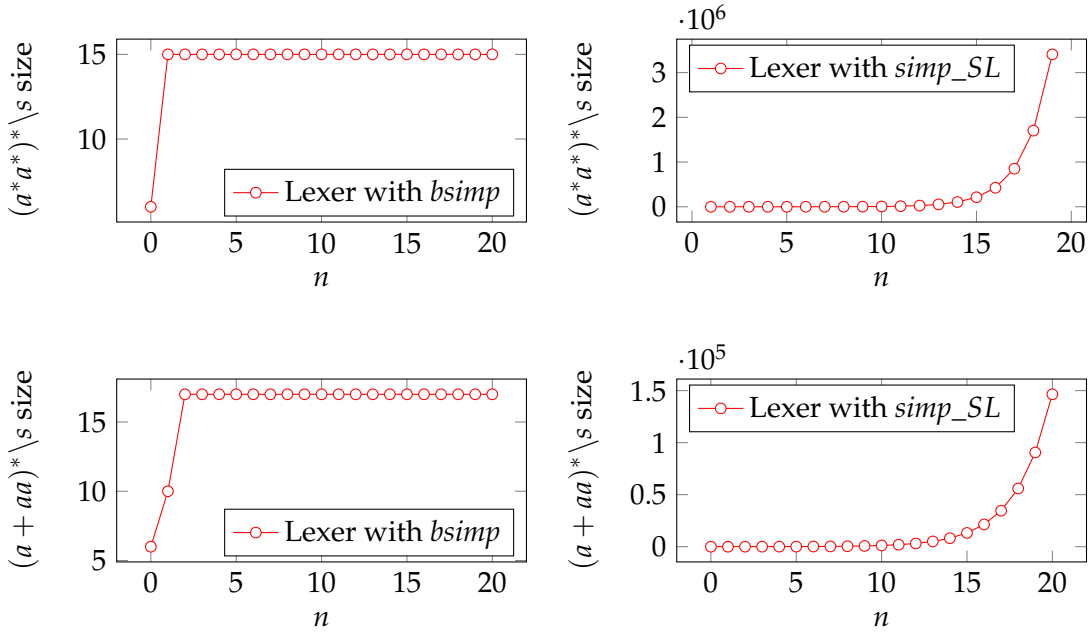
Example 3:

$$(a + aa)^* \xrightarrow{(\_)^\uparrow} (_Z a +_S aa)^* \xrightarrow{\backslash a} {}_Z\,(_Z\mathbf{1} +_S (\mathbf{1}a)) \cdot (_Z a +_S aa)^*$$
$$\xrightarrow{bsimp} {}_Z\,(_Z\mathbf{1} +_S a) \cdot (_Z a +_S aa)^* \xrightarrow{\backslash a} {}_Z$$
$$(\mathbf{0} +_S \mathbf{1}) \cdot (_Z a +_S aa)^* +_{ZZ} (_Z(_Z\mathbf{1} +_S (\mathbf{1}a)) \cdot (_Z a +_S aa)^*)$$
$$\xrightarrow{bsimp} {}_{ZS}\,(_Z a +_S aa)^* +_{ZZ} (_Z(_Z\mathbf{1} +_S a) \cdot (_Z a +_S aa)^*) \xrightarrow{bmkeps} ZSS \xrightarrow{decode\ (a+aa)^*} Stars\ [aa]$$

This algorithm keeps the regular expression size small, as we shall demonstrate with some examples in the next section.

5.2.4 Examples $(a + aa)^*$ and $(a^* \cdot a^*)^*$ After Simplification

Recall the previous $(a^*a^*)^*$ matching $\underbrace{aa\ldots a}_{\text{n as}}$ example where *simp_SL* could not prevent the fast growth. Now with *bsimp* the size is greatly reduced and stays below a constant no matter how long the input string is. We have similar trends for $(a + aa)^*$. This is shown in the graphs below.



Given the size difference, it is not surprising that our *blexer_simp* significantly outperforms *blexer_SLSimp* by Sulzmann and Lu. Indeed the intermediate derivatives of *blexer_simp* seem to stay below a constant bound. As promised we will use formal proofs to show that our speculation based on these experimental results indeed hold. In the next section we are going to establish that our simplification preserves the correctness of the algorithm.

5.3 Correctness of *blexer_simp*

In the *blexer*'s correctness proof, Ausaf et al. [14] did not directly derive the fact that *blexer* generates the POSIX value, but first proved that *blexer* generates the same result as *lexer*. Then they re-used the correctness of *lexer* to obtain theorem 2. For *blexer_simp* we build on Ausaf et al.'s result, by proving that *blexer_simp r s* produces the same output as *blexer r s*.

We first thought of directly re-using the proof techniques from *blexer*'s correctness proof. However we were not able to find a suitable modification to the key property 6 because simplification breaks things.

5.3.1 Why *Blexer*'s Proof Does Not Work

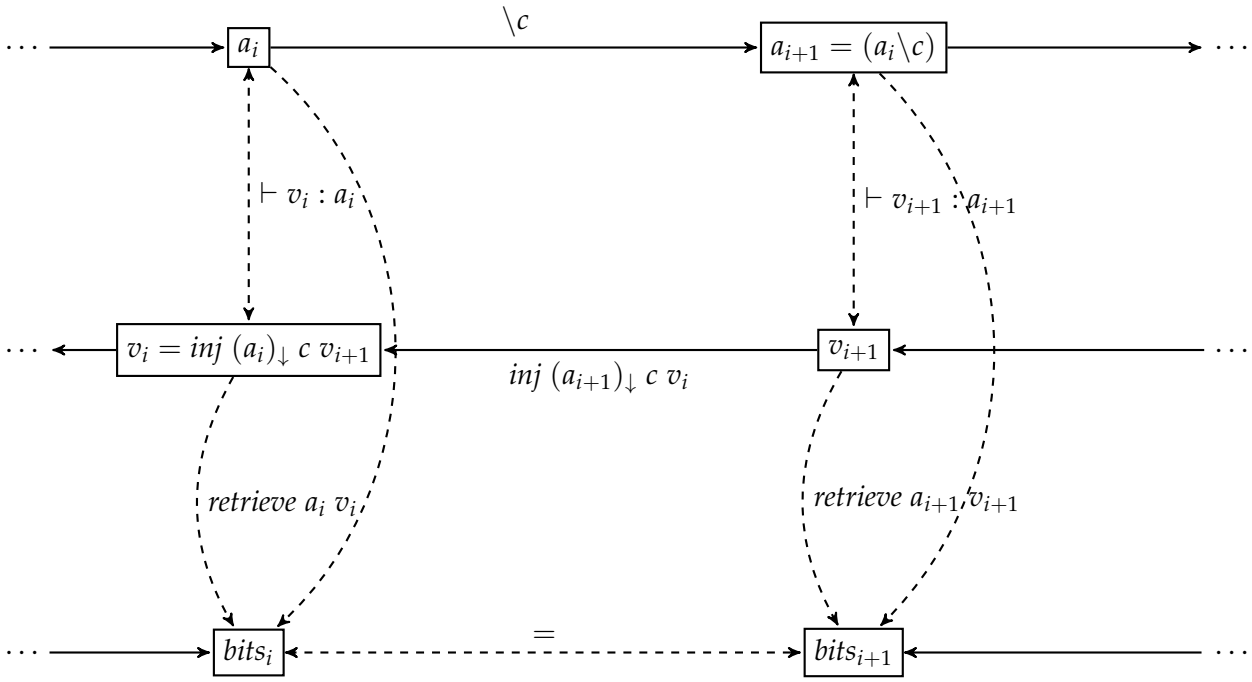
The fundamental reason is because lemma 6 does not hold anymore when simplifications are involved. In particular, the correctness theorem of *blexer* relies crucially on property 6 that says bitcodes can be retrieved from before and after the derivative, using values from after and before the injection:

$$\vdash v : ((a_{\downarrow})\backslash c) \implies \mathit{retrieve}\ (a\backslash c)\ v = \mathit{retrieve}\ a\ (\mathit{inj}\ (a_{\downarrow})\ c\ v) \tag{5.1}$$

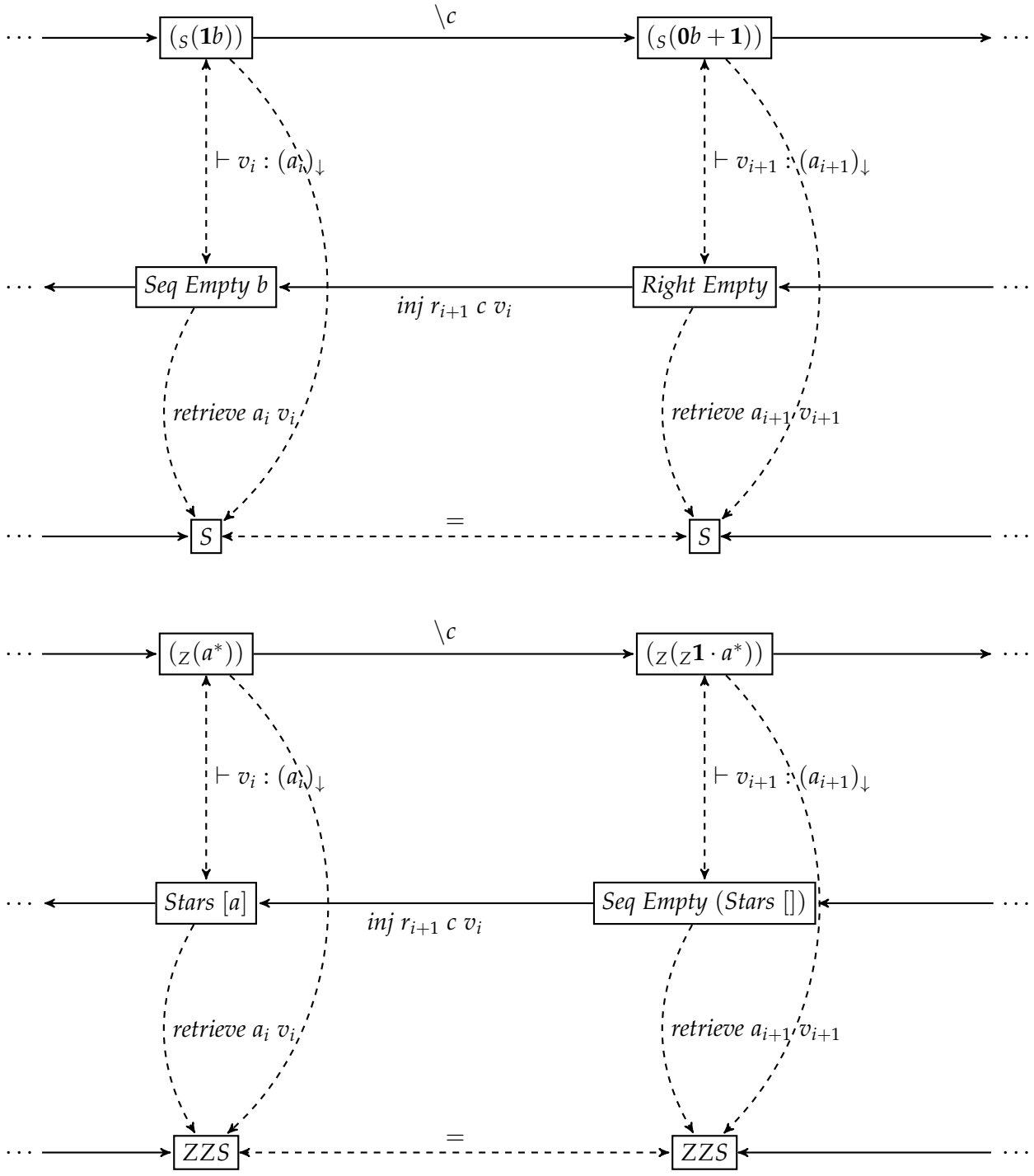
The pairs

$$(a,\ \mathit{inj}\ a_{\downarrow}\ c\ v) \text{ and } (a\backslash c,\ v)$$

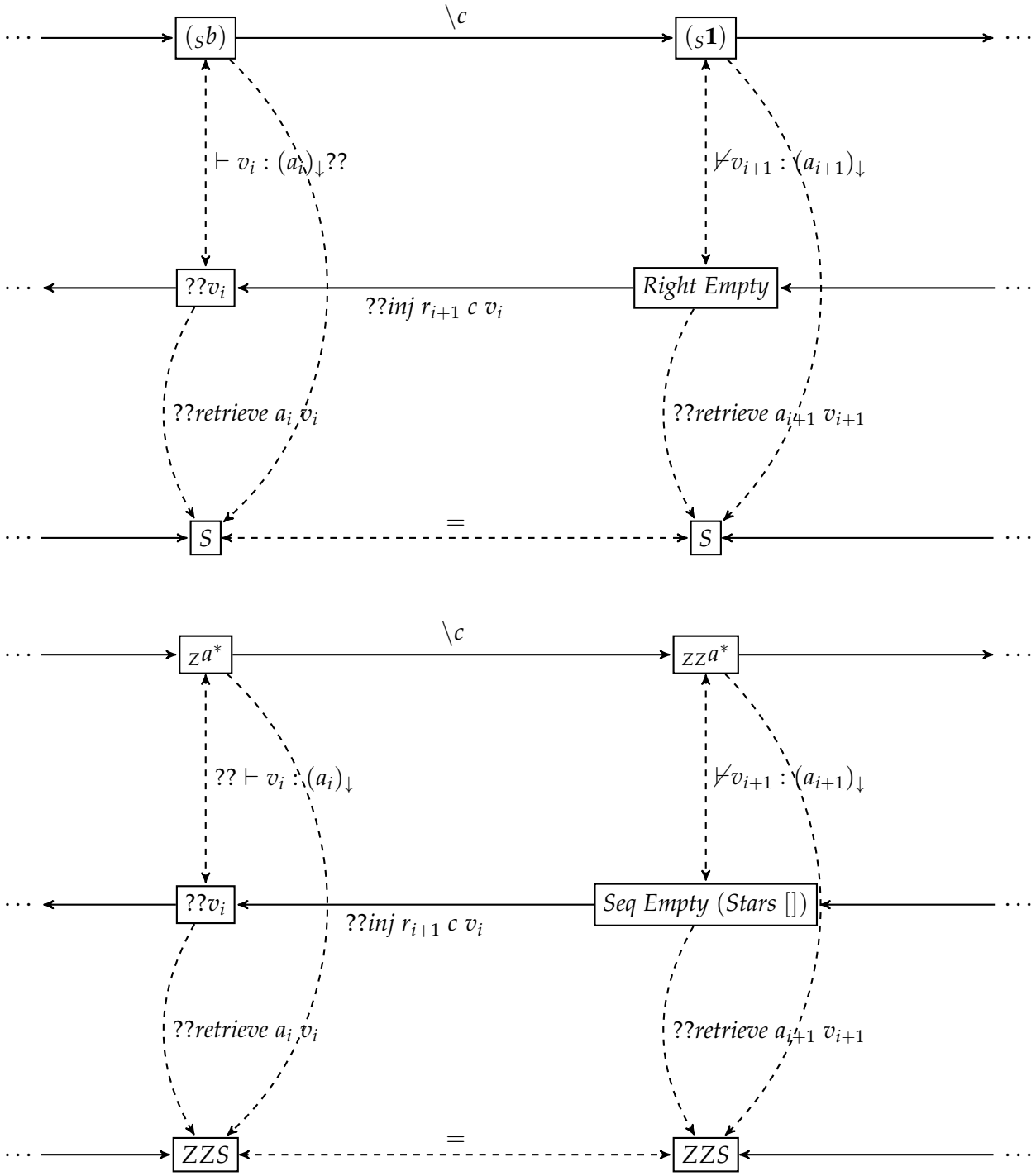
are from two consecutive steps in the intermediate derivatives. If we call these two steps' derivatives a_i and a_{i+1} and their inhabited POSX value v_i and v_{i+1}, then one can create a diagram showing their relation pictorially:



Here are some concrete instances of this correspondence for *blexer*:



As *blexer_simp* applies *bsimp* after each call to the derivatives function, it becomes a problem to maintain the same property. It is not clear how to inject back a character to a simplified value.



Previously *retrieve* works properly, but with *blexer_simp* it becomes impossible unless we re-design *retrieve* and *inj*. For instance, if we change the form of property 5.1 to adapt to the needs of *blexer_simp* the precondition becomes

$$\vdash v' : (bsimp\ (a\backslash c))_\downarrow,$$

and *inj* is in general not defined on the input $a_\downarrow$, c and v'. They may accidentally work, like for the first example, where we have $a =_S b$ and $v' = Empty$. For the second exmaple, we do not know what $inj\ a^*\ a\ (Stars\ [])$ should be. One might attempt to define a new injection like $inj'\ c^*\ c\ (Stars\ []) \stackrel{\text{def}}{=} Stars\ [c]$ to target this

specific case, however we have not yet found a generalisation from this to more complex star regular expressions. The retrieve function will not work either. It seems unclear what procedures needs to be used to create a new value $v_?$ such that

$$\vdash v_? : r \text{ and } \mathit{retrieve}\ r\ v_? = \mathit{retrieve}\ (\mathit{bsimp}\ (r\backslash c))\ v'$$

holds. Without *retrieve* the bridge from *code* v to *bmkeps* $(a\backslash s)$ no longer exists. Ausaf et al. [14] used something they call rectification functions to restore the original value from the simplified value. The idea is that simplification functions not only returns a regular expression, but also a rectification function

$$\mathit{simp}^{\mathit{rect}} : \mathit{Regex} \Rightarrow (\mathit{Value} \Rightarrow \mathit{Value}, \mathit{Regex})$$

that is recorded recursively, and then applied to the previous value to obtain the correct value for *inj* to work on. The recursive case of the lexer is defined as something like

$$\mathit{slexer}\ r\ (c::s) \stackrel{\text{def}}{=} \mathit{let}\ (\mathit{frect}, r_c) = \mathit{simp}^{\mathit{rect}}\ (r\backslash c)\ \ \mathit{in}\ \ \mathit{inj}\ r\ c\ (\mathit{frect}\ (\mathit{slexer}\ r_c\ s))$$

However this approach (including *slexer*'s correctness proof) only works without bitcodes, and it limits the kind of simplifications one can introduce. See the thesis by Ausaf [13] for details.

We were not able to use their idea for our simplification rules. Instead, we took another route that completely disposes of property 6, and prove a weakened inductive invariant instead. In the next section, we first explain why property 6's requirement is too strong, and suggest a few possible relaxation, which leads to our proof that seemed to us natural and effective.

5.3.2 Why Lemma 6's Requirement is too Strong

Consider the annotated regular expressions $a_i = (_{ZZ}x +_{ZS} y +_S x)$ and $a_{i+1} = (_{ZZ}\mathbf{1} + \mathbf{0} +_S \mathbf{1})$. We only need in the proof of *blexer* that for the POSIX value $v_i = \mathit{Left}\ (\mathit{Left}\ \mathit{Empty})$, the property

$$\mathit{retrieve}\ (_{ZZ}\mathbf{1} + \mathbf{0} +_S \mathbf{1})\ (\mathit{Left}\ (\mathit{Left}\ \mathit{Empty})) = \mathit{retrieve}\ (_{ZZ}x +_{ZS} y +_S x)\ (\mathit{Left}\ (\mathit{Left}\ \mathit{Char}\ x))$$

holds, and for *blexer_simp* the POSIX terms $_{ZZ}\mathbf{1}$ and $_{ZZ}x$ are present as well. Therefore their bitcodes can be extracted. However for the definitely non-POSIX value $v_i' = \mathit{Right}\ \mathit{Empty}$ the following equality

$$\mathit{retrieve}\ (_{ZZ}\mathbf{1} + \mathbf{0} +_S \mathbf{1})\ (\mathit{Right}\ \mathit{Empty}) = \mathit{retrieve}\ (_{ZZ}x +_{ZS} y +_S x)\ (\mathit{Right}\ (\mathit{Char}\ x))$$

also needs to hold. These values do not exist for *blexer_simp* asthey have been eliminated during the de-duplication procedure of our smplification. If we were to use *retrieve*, then we are stuck with a property that holds in *blexer* but does not have a counterpart in *blexer_simp*.

The inductive invariant 5.1 can be weakened by strengthening the precondition $\vdash v_i : r_i$ to $\exists s_i.\ (s_i, r_i) \rightarrow v_i$, namely that v_i must be a POSIX value. We tried this route but it did not work well since we need to use a similar technique as the rectification functions by Ausaf et al, and they can get very complicated with *bsimp*.

Another inductive invariant we considered was that

$$\mathit{bmkeps}\ a_i = \mathit{code}\ v_i,$$

namely one can extract the POSIX value using *bmkeps* rather than *retrieve*. But this condition is too weak such that one cannot get through the inductive step. For instance,

$$bmkeps\ (_Z\mathbf{1} +_{SZ} a^*) = Z = code\ Left(Seq\ a\ a)$$

holds. However this property is about the term $_Z\mathbf{1}$ alone. It cannot be used to deduce

$$bmkeps\ (_Z\mathbf{0} +_{SZZ} a^*) = SZZS = code\ Right(Stars\ [aa])$$

because it does not say anything about the second term $_{SZ}a^*$.

To summarise, we need a property that says POSIX values can be extracted from every intermediate step, and that property cannot be too strong to involve definitely non-POSIX terms, or too weak to exclude potentially POSIX terms. A natural idea we came up with was to define a rewriting relation from $a\backslash s$ to $a\backslash_{bsimps} s$. Such relations are sound with respect to POSIX rules because they only remove the second duplicate. We prove that if one term is reachable from another via the rewriting relation, then it will always output the same lexing information from *bmkeps*.

In the next section we first introduce the rewriting relation *rrewrite* (*rrewrite*) between two regular expressions, which stands for an atomic simplification. We then prove properties about this rewriting relation and its reflexive transitive closure. Finally we leverage these properties to show an equivalence between the results generated by *blexer* and *blexer_simp*.

5.3.3 The Rewriting Relation *rrewrite*($\rightsquigarrow$)

The idea of a single-step rewriting relation *rrewrite* is that the transition from r to $bsimp\ r$ can be broken down into smaller rewrite steps of the form:

$$r \rightsquigarrow^* bsimp\ r$$

where each rewrite step, written $\rightsquigarrow$, is an "atomic" simplification that is more or less similar to small-step operational semantics:

$$\frac{}{{}_{bs}\mathbf{0} \cdot r_2 \rightsquigarrow \mathbf{0}}\, S\mathbf{0}_l \qquad \frac{}{{}_{bs}r_1 \cdot \mathbf{0} \rightsquigarrow \mathbf{0}}\, S\mathbf{0}_r \qquad \frac{}{{}_{bs1}((_{bs2}\mathbf{1}) \cdot r) \rightsquigarrow fuse\ (bs_1 @ bs_2)\ r}\, S_1$$

$$\frac{r_1 \rightsquigarrow r_2}{{}_{bs}r_1 \cdot r_3 \rightsquigarrow_{bs} r_2 \cdot r_3}\, SL \qquad \frac{r_3 \rightsquigarrow r_4}{{}_{bs}r_1 \cdot r_3 \rightsquigarrow_{bs} r_1 \cdot r_4}\, SR$$

$$\frac{}{{}_{bs}\sum[] \rightsquigarrow \mathbf{0}}\, A0 \qquad \frac{}{{}_{bs}\sum[a] \rightsquigarrow fuse\ bs\ a}\, A1 \qquad \frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{{}_{bs}\sum rs_1 \rightsquigarrow rs_2}\, AL \qquad \frac{}{[] \overset{s}{\rightsquigarrow} []}\, LE$$

$$\frac{rs_1 \overset{s}{\rightsquigarrow} rs_2}{r :: rs_1 \overset{s}{\rightsquigarrow} r :: rs_2}\, LT \qquad \frac{r_1 \rightsquigarrow r_2}{r_1 :: rs \overset{s}{\rightsquigarrow} r_2 :: rs}\, LH \qquad \frac{}{\mathbf{0} :: rs \overset{s}{\rightsquigarrow} rs}\, L\mathbf{0}$$

$$\frac{}{{}_{bs}\sum(rs_1 :: rs_b) \overset{s}{\rightsquigarrow} ((map\ (fuse\ bs_1)\ rs_1) @ rsb)}\, LS$$

$$\frac{(a_1)_{\downarrow_r} = (a_2)_{\downarrow_r}}{rs_a @ [a_1] @ rs_b @ [a_2] @ rs_c \overset{s}{\rightsquigarrow} rs_a @ [a_1] @ rs_b @ rs_c}\, LD$$

FIGURE 5.4: The rewrite rules that generate simplified regular expressions in small steps: $r_1 \rightsquigarrow r_2$ is for bitcoded regular expressions and $rs_1 \overset{s}{\rightsquigarrow} rs_2$ for lists of bitcoded regular expressions. Interesting is the LD rule that allows copies of regular expressions to be removed provided a regular expression earlier in the list can match the same strings.

The rules LT and LH are for rewriting two regular expression lists such that one regular expression in the left-hand-side list is reachable in one step to the right-hand side's regular expression at the same position. This helps with defining the "context rule" AL. These inference rules at first glance seem just congruence relations for regular expressions that denote the same language. However in the context of getting POSIX values it is not immediately obvious why the relation will generate the same bitcodes, namely

$$r \rightsquigarrow^* r' \implies blexer\ r\ s = blexer\_simp\ r\ s.$$

Indeed if we add an inference rule like

$$\frac{(a_1)_{\downarrow_r} = (a_2)_{\downarrow_r}}{rs_a @ [a_1] @ rs_b @ [a_2] @ rs_c \overset{s}{\rightsquigarrow} rs_a @ rs_b @ [a_2] @ rs_c}\, KEEPRIGHT$$

then POSIX values will be lost in rewriting. We believe this rewriting system is terminating because the terms are all first-order, and each rewriting reduces the size of a regular expression. The rules are not confluent unless we define two terms to be equivalent if they only differ in the position of bitcodes, for example

$$_Z(_Z a +_S b) \text{ and } _{ZZ}a +_{ZS} b.$$

We leave the formal proof of these for future work.

The reflexive transitive closure of $\rightsquigarrow$ and $\overset{s}{\rightsquigarrow}$ are defined in the usual way:

$$\frac{}{r \rightsquigarrow^* r} \qquad \frac{}{rs \overset{s*}{\rightsquigarrow} rs} \qquad \frac{r_1 \rightsquigarrow^* r_2 \wedge\ r_2 \rightsquigarrow^* r_3}{r_1 \rightsquigarrow^* r_3} \qquad \frac{rs_1 \overset{s*}{\rightsquigarrow} rs_2 \wedge\ rs_2 \overset{s*}{\rightsquigarrow} rs_3}{rs_1 \overset{s*}{\rightsquigarrow} rs_3}$$

The main theorems we are going to prove for this chapter is that the rewriting relation commutes with derivatives:

$$r_1 \rightsquigarrow r_2 \implies (r_1 \backslash c) \rightsquigarrow^* (r_2 \backslash c)$$

And also, if two terms are reachable from one another via rewrites, then they produce the same bitcodes under *bmkeps*:

$$r \rightsquigarrow^* r' \ \ then \ \ bmkeps\ r = bmkeps\ r'$$

These two properties will serve as the bridge between *blexer* and *blexer_simp*. The decoding phase of both functions are the same, which means that if they receive the same bitcodes from *bmkeps*, then they generate the same values from *decode*. We will provide more details of these properties in the next sub-section.

5.3.4 Important Properties of $\rightsquigarrow$

First we list some basic facts about $\rightsquigarrow$, $\overset{s}{\rightsquigarrow}$, $\rightsquigarrow^*$ and $\overset{s*}{\rightsquigarrow}$. They can be usually solved with straightforward induction and lightweight automation in Isabelle. They are of quite similar nature and therefore grouped together.

Lemma 1. *The inference rules (5.4) we gave in the previous section have their "many-steps version":*

- $rs_1 \overset{s*}{\rightsquigarrow} rs_2 \implies {}_{bs}\sum rs_1 \rightsquigarrow^*_{bs} \sum rs_2$
- $r \rightsquigarrow^* r' \implies {}_{bs}\sum(r :: rs) \ \rightsquigarrow^* \ {}_{bs}\sum(r' :: rs)$
- *The rewriting in many steps property is composable in terms of the sequence constructor:*
  $r_1 \rightsquigarrow^* r_2 \implies {}_{bs}r_1 \cdot r_3 \rightsquigarrow^* {}_{bs}r_2 \cdot r_3 \quad$ *and* $\quad r_3 \rightsquigarrow^* r_4 \implies {}_{bs}r_1 \cdot r_3 \rightsquigarrow^*_{bs} r_1 \cdot r_4$
- *The rewriting in many steps properties $\rightsquigarrow^*$ and $\overset{s*}{\rightsquigarrow}$ is preserved under the function fuse:*
  $r_1 \rightsquigarrow^* r_2 \implies fuse\ bs\ r_1 \rightsquigarrow^* fuse\ bs\ r_2 \quad and\ rs_1 \overset{s}{\rightsquigarrow} rs_2 \implies map\ (fuse\ bs)\ rs_1 \overset{s*}{\rightsquigarrow} map\ (fuse\ bs)\ rs_2$

The inference rules of $\overset{s}{\rightsquigarrow}$ are defined in terms of the list cons operation. Now we show that they also hold w.r.t prepending and appending of lists.

- $rs_1 \overset{s}{\rightsquigarrow} rs_2 \implies rs@rs_1 \overset{s}{\rightsquigarrow} rs@rs_2$
- $rs_1 \overset{s*}{\rightsquigarrow} rs_2 \implies rs@rs_1 \overset{s*}{\rightsquigarrow} rs@rs_2 \ \ and \ \ rs_1@rs \overset{s*}{\rightsquigarrow} rs_2@rs$
- *The $\overset{s}{\rightsquigarrow}$ relation after appending a list becomes $\overset{s*}{\rightsquigarrow}$:*
  $rs_1 \overset{s}{\rightsquigarrow} rs_2 \implies rs_1@rs \overset{s*}{\rightsquigarrow} rs_2@rs$

In addition, we also prove some relations between $\rightsquigarrow^*$ and $\overset{s*}{\rightsquigarrow}$.

- $r_1 \rightsquigarrow^* r_2 \implies [r_1] \overset{s*}{\rightsquigarrow} [r_2]$
- $rs_3 \overset{s*}{\rightsquigarrow} rs_4 \wedge r_1 \rightsquigarrow^* r_2 \implies r_2 :: rs_3 \overset{s*}{\rightsquigarrow} r_2 :: rs_4$
- *If we can rewrite a regular expression in many steps to* $\mathbf{0}$*, then we can also rewrite any sequence containing it to* $\mathbf{0}$*:*
  $r_1 \rightsquigarrow^* \mathbf{0} \implies {}_{bs}r_1 \cdot r_2 \rightsquigarrow^* \mathbf{0}$

Proofs of these are omitted, and details can be found in [46].

With all the smaller lemmas above, we are ready to prove the more important properties:

- $r \rightsquigarrow^* r' \wedge bnullable\ r_1 \implies bmkeps\ r = bmkeps\ r'$.
- $r \rightsquigarrow^* bsimp\ r$.
- $r \rightsquigarrow r' \implies r\backslash c \rightsquigarrow^* r'\backslash c$.

Intuitively, the first property says we can extract the same bitcodes using *bmkeps* from the nullable components of two regular expressions r and r', if we can rewrite from one to the other in finitely many steps.

For convenience, we define a predicate for a list of regular expressions having at least one nullable regular expression:

$$bnullables\ rs \quad \overset{\text{def}}{=} \quad \exists r \in rs.\ bnullable\ r$$

And similarly, *bmkepss* which extracts the bit-codes on the first *bnullable* element in a list:

$$\begin{array}{lcl} bmkepss\ [] & \overset{\text{def}}{=} & [] \\ bmkepss\ r :: rs & \overset{\text{def}}{=} & if\ (bnullable\ r)\ \ then\ \ bmkeps\ r\ else\ \ bmkepss\ rs \end{array}$$

Lemma 2. *The rewriting relation* $\rightsquigarrow$ *preserves (b)nullability:*

- *If* $r_1 \rightsquigarrow r_2$*, then* $bnullable\ r_1 = bnullable\ r_2$
- *If* $rs_1 \overset{s}{\rightsquigarrow} rs_2$ *then* $bnullables\ rs_1 = bnullables\ rs_2$
- $r_1 \rightsquigarrow^* r_2 \implies bnullable\ r_1 = bnullable\ r_2$

If both regular expressions in a rewriting relation are nullable, then they produce the same bitcodes:

- $r_1 \rightsquigarrow r_2 \implies (bnullable\ r_1 \wedge bnullable\ r_2 \implies bmkeps\ r_1 = bmkeps\ r_2)$
- *and* $rs_1 \overset{s}{\rightsquigarrow} rs_2 \implies (bnullables\ rs_1 \wedge bnullables\ rs_2 \implies bmkepss\ rs_1 = bmkepss\ rs2)$

Proofs to these smaller lemmas are omitted. As they can be done with relative ease by inducting on the case of $\rightsquigarrow$ and $\overset{s}{\rightsquigarrow}$.

Now we are ready for the key lemma saying that one can extract the same bitcodes using *bmkeps* for regular expressions that rewrite to each other in many steps:

Lemma 3. *If* $r \rightsquigarrow^* r'$ *and* *bnullable* r, *then* *bmkeps* $r =$ *bmkeps* r'

Proof. We induct on the cases that lead to $\rightsquigarrow^*$. For $r' = r$, then clearly *bmkeps* $r =$ *bmkeps* r'. Now for the inductive case, the assumption is that $r \rightsquigarrow^* r'$, *bnullable* r and *bmkeps* $r =$ *bmkeps* r' holds (by point 1 and 3 of lemma 2). Now for $r' \rightsquigarrow r''$, we know that both *bnullable* r' and *bnullable* r'' hold. Therefore *bmkeps* $r' =$ *bmkeps* r'' (by point 4 of lemma 2). □

Now we prove that $\rightsquigarrow$ is a complete set of rules for *bsimp*, in other words, one can describe the simplification involved in *bsimp* using all steps from $\rightsquigarrow$: $r \rightsquigarrow^*$ *bsimp* r. For this we need to prove that *bsimp*'s helper functions such as *distinctBy* and *flts* can all be described by $\overset{s*}{\rightsquigarrow}$ and $\rightsquigarrow^*$.

Lemma 4.

- *The first lemma for the completeness of* $\rightsquigarrow$ *is to prove is a more general version of* $rs_1 \rightsquigarrow^*$ *distinctBy* rs_1 ϕ*: For a list made of two parts* $rs_1@rs_2$*, one can throw away the duplicate elements in* rs_2*, as well as those that have appeared in* rs_1. $rs_1@rs_2 \overset{s*}{\rightsquigarrow} (rs_1@(distinctBy\ rs_2\ \ rerase\ \ (map\ \ rerase\ \ rs_1)))$
- *(A corollary of the above property by setting* rs_2 *to be the empty list.)* $rs_1 \overset{s*}{\rightsquigarrow}$ *distinctBy* rs_1 ϕ.
- *Similarly the flatten function flts describes a reduct of* $\overset{s*}{\rightsquigarrow}$ *as well:*

$$rs \overset{s*}{\rightsquigarrow} flts\ rs$$

- *The function* $bsimp_{ALTS}$ *preserves can also be described by* $\rightsquigarrow$: ${}_{bs}\sum rs \rightsquigarrow^* bsimp_{ALTS}\ {}_{bs}\ rs$

Now we can prove that $\rightsquigarrow$ is complete in the sense that it describes all possible simplifications that happen in *bsimp*:

Theorem 3. $r \rightsquigarrow^*$ *bsimp* r

Proof. By an induction on r. The most involved case is the alternative, where we use part 2, 3 and 4 of lemma 4 to justify the following rewriting steps:

$$\begin{array}{lll} rs & \overset{s*}{\rightsquigarrow} & map\ bsimp\ rs \\ & \overset{s*}{\rightsquigarrow} & flts\ (map\ bsimp\ rs) \\ & \overset{s*}{\rightsquigarrow} & distinctBy\ (flts\ (map\ bsimp\ rs))\ rerase\ \phi \end{array}$$

Using this we can derive the following rewrite sequence:

$$\begin{array}{lll} r & = & {}_{bs}\sum rs \\ & \rightsquigarrow^* & bsimp_{ALTS}\ bs\ rs \\ & \rightsquigarrow^* & bsimp_{ALTS}\ bs\ (map\ bsimp\ rs) \\ & \rightsquigarrow^* & bsimp_{ALTS}\ bs\ (flts\ (map\ bsimp\ rs)) \\ & \rightsquigarrow^* & bsimp_{ALTS}\ bs\ (distinctBy\ (flts\ (map\ bsimp\ rs))\ rerase\ \phi) \\ & \rightsquigarrow^* & bsimp\ r \end{array}$$

The rest of the cases are routine and can be proven with straightforward induction and automation. □

Now we are going to prove the central theorem that leads to the correctness of *blexer_simp*: $r_1 \rightsquigarrow^* r_2 \implies r_1\backslash c \rightsquigarrow^* r_2\backslash c$ This property justifies why we are able to interleave derivatives and simplifications and still able to get the right answer.

The rewrite relation $\rightsquigarrow$ changes into $\rightsquigarrow^*$ after derivatives are taken on both sides:

Lemma 5.

- *If* $r_1 \rightsquigarrow r_2$, *then* $r_1\backslash c \rightsquigarrow^* r_2\backslash c$
- *If* $rs_1 \overset{s}{\rightsquigarrow} rs_2$, *then map* $(\_\backslash c)\ rs_1 \overset{s*}{\rightsquigarrow} map\ (\_\backslash c)\ rs_2$

Proof. For part one, we induct on the inference rules of $\rightsquigarrow$. For each inference rule we do a case analysis on r_1. There are a few dozen cases, each solvable using automation. The second part is a corollary of the first. □

Now we can prove the central theorem as an immediate corollary:

Theorem 4 (Central theorem of commuting derivatives and simplifications)**.**

- $r_1 \rightsquigarrow^* r_2 \implies r_1\backslash c \rightsquigarrow^* r_2\backslash c$
- $a\backslash s \rightsquigarrow^* a\backslash_{bsimps} s$

We omit the proof as they follow straightforwardly from lemma 5 and theorem 3.

5.3.5 Main Theorem

Now with theorem 4 in place we are ready for the main result of this chapter:

Corollary 1 (Correctness of *blexer_simp*)**.**

- $$blexer\ r\ s = blexer\_simp\ r\ s$$
- $$(r, s) \rightarrow v \ \ iff \ \ blexer\_simp\ r\ s = Some\ v$$
$$\nexists v.\ (r, s) \rightarrow v \ \ iff \ \ blexer\_simp\ r\ s = None.$$

Proof. We first consider the case where $s \in L\ r$. Then *blexer r s* is equal to

$$decode\ (bmkeps\ (r^\uparrow\backslash s))\ r.$$

Similarly *blexer_simp r s* is equal to

$$decode\ (bmkeps\ (r^\uparrow\backslash_{bsimps} s))\ r.$$

We prove that these two terms are equal by proving that the two bitcode input are equal:

$$bmkeps\ (r^\uparrow\backslash s) = bmkeps\ (r^\uparrow\backslash_{bsimps} s).$$

Let $a = r^\uparrow$, we know that

$$a\backslash s \rightsquigarrow^* a\backslash_{bsimps} s.$$

by theorem 4. We also know that

$$bnullable\ (a\backslash s)$$

because $s \in L\ r$, which means

$$bmkeps\ (a\backslash s) = bmkeps\ (a\backslash_{bsimps} s)$$

by lemma 3. This concludes the $s \in L\ r$ case. When $s \notin L\ r$, we still have that

$$a\backslash s \rightsquigarrow^* a\backslash_{bsimps} s.$$

We also know that $bnullable\ (a\backslash s) = False$. Therefore, $bnullable\ (a\backslash_{bsimps} s) = False$ as well. Therefore both lexers will return *None*. The second part is a corollary of the first proposition. □

5.3.6 Comments on the Proof

The rewriting relation method we came up with was natural and intuitive, however we tried many things that did not work before arriving at this simple solution. Directly proving $blexer\ r\ s = blexer\_simp\ r\ s$ by a structural induction on r does not work, because $bsimp\ \sum rs \neq \sum(map\ bsimp\ rs)$–both *flts* and $bsimp_{ALTS}$ will alter the structure of an alternative regular expression, and it is not predictable which structure $bsimp\ \sum rs$ will end up in. Therefore we cannot use the inductive hypothesis.

We also attempted to re-use the argument in lemma 6. We tried the Ausaf et al.'s rectification function, however with *flts* the rectification got soon very complicated and unmanageable.

We also tried to prove something like

$$bsimp\ \ (a\backslash_{bsimps} s) \cong bsimp\ \ (a\backslash s),$$

but a direct equality does not hold. A counterexample is

$$a = [(_Z 1 +_S c) \cdot [bb \cdot (_Z 1 +_S c)]] \ \text{ and } \ s = bb.$$

Then we would have

$$bsimp\ \ (a\backslash s) = {}_{[]}(_{ZZ}\mathbf{1} +_{ZS} c)$$

whereas

$$bsimp\ \ (a\backslash_{bsimps} s) = {}_Z(_Z\mathbf{1} +_S c).$$

Unfortunately, if we apply *bsimp* differently we will always have this discrepancy. This is due to the $map\ (fuse\ bs)\ as$ operation happening at different times. This requires us to define a notion of *canonical* form, and proving that both $bsimp\ \ (a\backslash_{bsimps} s)$ and $bsimp\ \ (a\backslash s)$ can be turned into the canonical form by just reordering the bits, for example pushing the bits of ${}_Z(_Z\mathbf{1} +_S c)$ in to get ${}_{[]}(_{ZZ}\mathbf{1} +_{ZS} c)$. That is how we thought of using rewriting relations: it seems natural to define the bits-moving operations as rewriting relations, for instance by mandating that ${}_{bs}\sum rs \rightsquigarrow \sum map\ (fuse\ bs)\ rs$ and define the symmetric transitive closure $\rightsquigarrow^*$ so that we can prove something like $bsimp\ \ (a\backslash_{bsimps} s) \rightsquigarrow^* r_{canonical form}$ and $bsimp\ \ (a\backslash s) \rightsquigarrow^* r_{canonical form}$. and use the canonical form $r_{canonical form}$ as the bridge between $bsimp\ \ (a\backslash_{bsimps} s)$ and $bsimp\ \ (a\backslash s)$. Soon we realised this approach can be further simplified and that is how we arrived at the current proof approach.

The rewriting relation $\rightsquigarrow^*$ allows us to ignore this discrepancy and view the expressions

$$[](ZZ\mathbf{1} +_{ZS} c)$$
and
$$Z(Z\mathbf{1} +_{S} c)$$

as equal because they were both re-written from the same expression.

The inference rules given in 5.4 are by no means final. One could come up with new rules by making use of more autamaton theory and Kleene algebra, for example by the rule

$$\frac{}{SEQr_1 \cdot (SEQr_1 \cdot r_3) \rightarrow SEQs[r_1, r_2, r_3].}\ SEQs$$

However this does not fit with the proof technique of our main theorem, but seem to not violate the POSIX property.

Having established the correctness of our *blexer_simp*, in the next chapter we shall prove that with our *bsimp* function, for a given r, the derivative size is always finitely bounded by a constant.

Chapter 6

A Formal Proof That *Blexer_simp* will not Grow Unbounded

In this chapter we prove for each r a bound in terms of the size of the calculated derivatives: given a regular expression r, let $a = r^{\uparrow}$. There exists a constant integer $C :: "int"$ (which depends on r), such that for any string s our algorithm *blexer_simp*'s intermediate derivatives' sizes are bounded by C.

$$[\![a \backslash_{bsimps} s]\!] \leq C$$

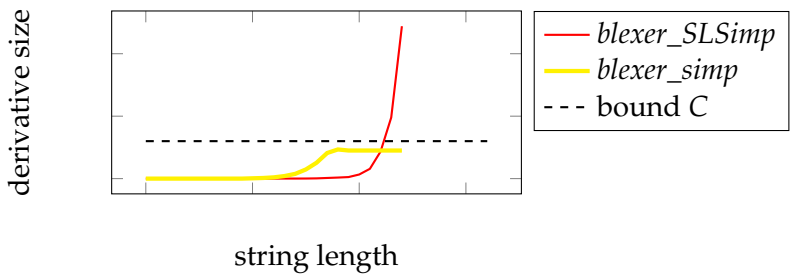
The size ($[\![\_]\!]$) of an annotated regular expression is defined in terms of the number of nodes in its tree structure (its recursive definition is given in the next page). Let *UNIV* be the set of all possible strings in our alphabet. This result can also be expressed as

$$\max\{[\![a \backslash_{bsimps} s]\!] \mid s \in UNIV\} \leq C.$$

Note that there are infinitely many terms in the set $\{a \backslash_{bsimps} s \mid s \in UNIV\}$ because the derivatives have different bit-annotations. We believe this size bound is important in the context of POSIX lexing because

- It is a stepping stone towards the goal of eliminating "catastrophic backtracking" once and for all. The derivative-based lexing algorithm attempts to avoid backtracking by a trade-off between space and time. Derivatives saves different matching possibilities as sub-expressions and traverse those during future derivatives. If such derivatives grow exponentially fast then we would still end up with exponential runtime. Having a constant-size rather than ever-increasing data structure gives us confidence that *blexer_simp* will not be unpredictably slow.
- The bound is universal for a given regular expression, which is an advantage over work which only gives empirical evidence on some test cases.

The bound plotted on a graph together with *blexer_SLSimp* and *blexer_simp*'s derivative sizes looks like:



We then extend our *blexer_simp* to support bounded repetitions ($r^{\{n\}}$). We update our formalisation of the correctness and finiteness properties to include this new construct, demonstrating the extensibility and generality of our proof method. Being able to handle bounded repetitions is nice because often regex engines are not very good at dealing with them. For instance, running the lexer against $a^{100}5$ and a string with 50000 a's costs the Verbatim++ [33] lexer over 5 minutes, but only a few seconds with our extracted Scala code. See [5] for a more detailed comparison with the Verbatim++ lexer, which is a verified lexer based on DFAs.

It is also possible to prove that the internal derivatives of *blexer_SLSimp* do grow unbounded, but in the time frame we only have experimental data showing the exponential growth trend.

The proofs in this chapter can be seen as a continuation of the proofs about the rewriting relations introduced in chapter 5. We prove more properties about them and also extend and use them to prove more properties about *blexer_simp*. This shows that our rewriting relation approach is a general and reusable proof technique.

In the first section we describe in more detail what the finite bound means in our algorithm and why the size of the internal data structures of a typical derivative-based lexer such as Sulzmann and Lu's needs formal treatment.

6.1 Formalising Size Bound of Derivatives

We first define what we mean by size.

$$\begin{array}{lcl}
[\![_{bs}\mathbf{1}]\!] & \stackrel{\text{def}}{=} & 1 \\
[\![\mathbf{0}]\!] & \stackrel{\text{def}}{=} & 1 \\
[\![_{bs}a_1 \cdot a_2]\!] & \stackrel{\text{def}}{=} & [\![a_1]\!] + [\![a_2]\!] + 1 \\
[\![_{bs}\mathbf{c}]\!] & \stackrel{\text{def}}{=} & 1 \\
[\![_{bs}\sum as]\!] & \stackrel{\text{def}}{=} & (sum\ (map\ ([\![_]\!])\ as)) + 1 \\
[\![_{bs}a^*]\!] & \stackrel{\text{def}}{=} & [\![a]\!] + 1.
\end{array}$$

The size of a derivative is defined by the number of nodes in the tree, and the bitcodes in our case do not count. This is the main reason why we are reluctant to declare that we have a fully formalised linear time complexity result–as the input string gets longer the bitcodes grow in proportional to it, and it is not obvious why all operations involving bitcodes are constant. We leave this for future work.

In our lexer (*blexer_simp*), we take an annotated regular expression as input, and repeatedly take derivative of and simplify it. Each time a derivative is taken, the regular expression might grow. However, the simplification that is immediately afterwards will often shrink it so that the overall size of the derivatives stays relatively small. This intuition is depicted by the relative size change between the black and blue nodes in figure 6.1. After *bsimp* the node shrinks. Our proof states that all the blue nodes stay below a size bound C determined by the input a.

Sulzmann and Lu's assumed a similar picture of their algorithm, even though it did not work as they expected. They tested out the run time of their lexer on particular examples such as $(a + b + ab)^*$ and made the speculation that their algorithm is linear w.r.t to the input. With our mechanised proof, we avoid this type of unintentional generalisation.

Before delving into the details of the formalisation, we are going to provide an overview of it in the following subsection.

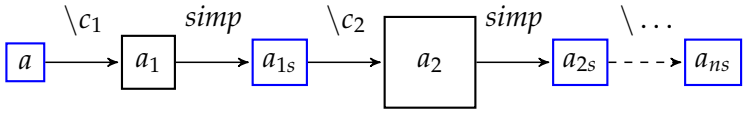


FIGURE 6.1: Regular expression size change during our *blexer_simp* algorithm

6.1.1 Overview of the Proof

For all discussions of this Chapter we will only use un-annotated regular expressions because bitcodes do not affect the size of a derivative. We will define the un-annotated version of functions like *bsimp* and $\backslash_{bsimps}$, but for the moment let's pretend we have them and call them *simp* and $\backslash_{simps}$ (More on this in section 6.2.1). The most important idea in this chapter is what we call the "closed forms" of regular expression derivatives with respect to strings. Very roughly speaking it expresses a regular expression r's (simplified) derivative w.r.t s as a list of derivative terms,

$$r \backslash_{simps} s \cong (\sum r_1 + r_2 + r_3 + \ldots + r_n),$$

where each term r_i is of the form $simp\ (r' \backslash s')$ or $simp\ (r' \backslash s') \cdot r''$, with r', r'' being a child expression of r or r itself, and s' being a sublist of s. The congruence relation means that LHS and RHS are eqaul up to simplifications. The closed forms are precise and formalised versions of this intuition. A few simple examples for this are:

$$(a^*a^*)^* \backslash_{simps} aaaaaaa = (a^*a^* + a^*) \cdot (a^*a^*)^* \cong \sum[(a^*a^*)^* \backslash a],$$

$$(a + aa)^* \backslash_{simps} aaa = (\mathbf{1} + a) \cdot (a + aa)^* \cong \sum[(a + aa)^* \backslash a],$$

$$(a + aa)^* \backslash_{simps} aaaa = (a + aa)^* + (\mathbf{1} + a) \cdot (a + aa)^* \cong$$

$$\sum[(a + aa)^* \backslash aa,\ (a + aa)^* \backslash a],$$

and

$$(aba + ab + a)^* \backslash_{simps} ababa = (aba + ab + a)^* + (ba + b + \mathbf{1}) \cdot (aba + ab + a)^*$$

$$\cong \sum[(aba + ab + a)^* \backslash aba,\ (aba + ab + a)^* \backslash a].$$

6.2 The *Rrexp* Datatype

The first step is to define *rrexp*s. They are regular expressions without bitcodes but allows a list of children expressions for alternatives, which allows a more convenient size bound proof. The datatype definition of the *rrexp*, called *r-regular expressions*, was initially defined in the last chapter at section 5.3. The reason for the prefix r is to make a distinction with basic regular expressions we introduced in chapter 3. We provide here again the definition to make this chapter self-contained as the proofs we introduce require this type:

$$rrexp ::= \mathbf{0}_r \mid \mathbf{1}_r \mid \mathbf{c}_r \mid r_1 \cdot r_2 \mid \sum rs \mid r^*$$

The size of an r-regular expression is written $[\![r]\!]_r$, whose definition mirrors that of an annotated regular expression.

$$
\begin{array}{lcl}
[\![_{bs}\mathbf{1}]\!]_r & \stackrel{\text{def}}{=} & 1 \\
[\![\mathbf{0}]\!]_r & \stackrel{\text{def}}{=} & 1 \\
[\![_{bs}r_1 \cdot r_2]\!]_r & \stackrel{\text{def}}{=} & [\![r_1]\!]_r + [\![r_2]\!]_r + 1 \\
[\![_{bs}\mathbf{c}]\!]_r & \stackrel{\text{def}}{=} & 1 \\
[\![_{bs}\sum as]\!]_r & \stackrel{\text{def}}{=} & (sum\ (map\ ([\![\_]\!]_r)\ as)) + 1 \\
[\![_{bs}a^*]\!]_r & \stackrel{\text{def}}{=} & [\![a]\!]_r + 1.
\end{array}
$$

The r in the subscript of $[\![]\!]_r$ is to differentiate with the same operation for annotated regular expressions. Similar subscripts will be added for operations like $()_{\downarrow_r}$:

$$
\begin{array}{lcl}
(\mathbf{0})_{\downarrow_r} & \stackrel{\text{def}}{=} & \mathbf{0}_r \\
(_{bs}\mathbf{1})_{\downarrow_r} & \stackrel{\text{def}}{=} & \mathbf{1}_r \\
(_{bs}\mathbf{c})_{\downarrow_r} & \stackrel{\text{def}}{=} & \mathbf{c}_r \\
(_{bs}r_1 \cdot r_2)_{\downarrow_r} & \stackrel{\text{def}}{=} & (r_1)_{\downarrow_r} \cdot (r_2)_{\downarrow_r} \\
(_{bs}\sum as)_{\downarrow_r} & \stackrel{\text{def}}{=} & \sum map\ (\_)_{\downarrow_r}\ as \\
(_{bs}a^*)_{\downarrow_r} & \stackrel{\text{def}}{=} & (a)^*_{\downarrow_r}
\end{array}
$$

6.2.1 Why a New Datatype?

Originally the erase operation $(\_)_\downarrow$ was used by Ausaf et al. in their proofs related to *blexer*. This function was not part of the lexing algorithm, and the sole purpose was to bridge the gap between the r (un-annotated) and *arexp* (annotated) regular expression datatypes so as to leverage the correctness theorem of *lexer*.For example, lemma 6 uses *erase* to convert an annotated regular expression a into a plain one so that it can be used by *inj* to create the desired value $inj\ (a)_\downarrow\ c\ v$.

Ideally *erase* should only remove the auxiliary information not related to the structure–the bitcodes. However there exists a complication where the alternative constructors have different arity for *arexp* and r:

$$
\begin{array}{lcl}
r & ::= & \ldots \mid (\_ + \_) :: "rexp \Rightarrow rexp \Rightarrow rexp" \mid \ldots \\
arexp & ::= & \ldots \mid (\Sigma\_) :: "arexp\ list \Rightarrow arexp" \mid \ldots
\end{array}
$$

To convert between the two *erase* has to recursively disassemble a list into nested binary applications of the $(\_ + \_)$ operator, handling corner cases like empty or singleton alternative lists:

$$
\begin{array}{lcl}
(_{bs}\sum[])_\downarrow & \stackrel{\text{def}}{=} & \mathbf{0} \\
(_{bs}\sum[a])_\downarrow & \stackrel{\text{def}}{=} & a \\
(_{bs}\sum a_1 :: a_2)_\downarrow & \stackrel{\text{def}}{=} & (a_1)_\downarrow + (a_2)_\downarrow \\
(_{bs}\sum a :: as)_\downarrow & \stackrel{\text{def}}{=} & a_\downarrow + (erase\ _{[]}\sum as)
\end{array}
$$

These operations inevitably change the structure and size of an annotated regular expression. This means that if we define the size of a basic plain regular expression in the natural way:

$$\begin{array}{lcl}
[\![\mathbf{1}]\!]_p & \stackrel{\text{def}}{=} & 1 \\
[\![\mathbf{0}]\!]_p & \stackrel{\text{def}}{=} & 1 \\
[\![r_1 + r_2]\!]_p & \stackrel{\text{def}}{=} & [\![r_1]\!]_p + [\![r_2]\!]_p + 1 \\
[\![\mathbf{c}]\!]_p & \stackrel{\text{def}}{=} & 1 \\
[\![r_1 \cdot r_2]\!]_p & \stackrel{\text{def}}{=} & [\![r_1]\!]_p \ + [\![r_2]\!]_p + 1 \\
[\![a^*]\!]_p & \stackrel{\text{def}}{=} & [\![a]\!]_p + 1,
\end{array}$$

then the property

$$[\![a]\!] \stackrel{?}{=} [\![a_{\downarrow}]\!]_p$$

does not hold. For example, $a_1 = \sum_Z [x]$ has size 2, but $(a_1)_{\downarrow} = x$ only has size 1. Another example is

$$a_2 = (_{ZZ}a +_{ZS} b +_S c).\ [\![a_2]\!] = 4, \text{ but } [\![(a_2)_{\downarrow}]\!]_p = [\![(a + (b + c))]\!]_p = 5$$

Bounds we obtain for *rrexp* does not translate into a bound for *arexp*–it might be too low. It might be higher but we do not know for sure. One might be able to prove an inequality such as $[\![a]\!] \leq [\![a_{\downarrow}]\!]_p + C$ and then get a bound on a from a bound on $a_{\downarrow}$ by adding the constant C, but we found our approach more straightforward. That leads to us defining *rrexp* whose size does not change during erase:

$$\begin{array}{lcl}
rrexp & ::= & \ldots \mid (\sum_) \ :: \ "rrexp\ list \Rightarrow rrexp" \mid \ldots \\
(_{bs}\sum as)_{\downarrow_r} & \stackrel{\text{def}}{=} & \sum map\ (_)_{\downarrow_r}\ as
\end{array}$$

This ensures that any bound on *rrexp* also applies to *arexp* (formalised in lemma 6).

6.2.2 Functions for R-regular Expressions

The downside of our approach is that we need to redefine several functions for *rrexp*. In this section we shall define the r-regular expression version of *bder*, and *bsimp* related functions. We use r as the prefix or subscript to differentiate with the bitcoded version. The derivative operation for an r-regular expression is

$$\begin{array}{lcl}
(\mathbf{0})\,\backslash_r c & \stackrel{\text{def}}{=} & \mathbf{0} \\
(\mathbf{1})\,\backslash_r c & \stackrel{\text{def}}{=} & \textit{if } c = d \textit{ then } \mathbf{1} \textit{ else } \mathbf{0} \\
(\sum\ rs)\,\backslash_r c & \stackrel{\text{def}}{=} & \sum\ (map\ (_\backslash_r c)\ rs) \\
(r_1 \cdot r_2)\,\backslash_r c & \stackrel{\text{def}}{=} & \textit{if } (rnullable\, r_1) \\
 & & \textit{then } \sum [(r_1\,\backslash_r c) \cdot r_2, \\
 & & \qquad ((r_2\,\backslash_r c))] \\
 & & \textit{else } (r_1\,\backslash_r c) \cdot r_2 \\
(r^*)\,\backslash_r c & \stackrel{\text{def}}{=} & (r\,\backslash_r c) \cdot (_{[]}r^*))
\end{array}$$

where we omit the definition of *rnullable*. The generalisation from the derivatives w.r.t a character to derivatives w.r.t strings is given as

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

The function *distinctBy* for r-regular expressions does not need a function checking equivalence because there are no bit annotations. Therefore we have

$$\begin{array}{lcl} rdistinct\ []\ rset & \stackrel{\text{def}}{=} & [] \\ rdistinct\ r :: rs\ rset & \stackrel{\text{def}}{=} & if(r \in rset)\ then\ rdistinct\ rs\ rset \\ & & else\ r :: rdistinct\ rs\ (rset \cup \{r\}) \end{array}$$

With *rdistinct* in place, the flatten function for *rrexp* is as follows:

$$\begin{array}{lcll} rflts\ (\sum as) :: as' & \stackrel{\text{def}}{=} & as\ @\ rflts\ as' & \\ rflts\ \mathbf{0} :: as' & \stackrel{\text{def}}{=} & rflts\ as' & \\ rflts\ a :: as' & \stackrel{\text{def}}{=} & a :: rflts\ as' & (\text{otherwise}) \end{array}$$

The function $rsimp_{ALTS}$ corresponds to $bsimp_{ALTS}$:

$$\begin{array}{lcl} rsimp_{ALTS}\ nil & \stackrel{\text{def}}{=} & \mathbf{0}_r \\ rsimp_{ALTS}\ r :: nil & \stackrel{\text{def}}{=} & r \\ rsimp_{ALTS}\ rs & \stackrel{\text{def}}{=} & \sum rs \end{array}$$

Similarly, we have $rsimp_{SEQ}$ which corresponds to $bsimp_{SEQ}$:

$$\begin{array}{lcl} rsimp_{SEQ}\ \mathbf{0}_r\ \_ & = & \mathbf{0}_r \\ rsimp_{SEQ}\ \_\ \mathbf{0}_r & = & \mathbf{0}_r \\ rsimp_{SEQ}\ \mathbf{1}_r \cdot r_2 & \stackrel{\text{def}}{=} & r_2 \\ rsimp_{SEQ}\ r_1 r_2 & \stackrel{\text{def}}{=} & r_1 \cdot r_2 \end{array}$$

and get *rsimp* and $\_\backslash_{rsimps}\_$:

$$\begin{array}{lcl} rsimp\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & rsimp_{SEQ}\ (rsimp\ r_1)\ (rsimp\ r_2) \\ rsimp\ ({}_{bs}\sum rs) & \stackrel{\text{def}}{=} & rsimp_{ALTS}\ bs\ (rdistinct\ (rflts(map\ rsimp\ rs))\ rerase\ \varnothing) \\ rsimp\ r & \stackrel{\text{def}}{=} & r \quad otherwise \end{array}$$

$$r\backslash_{rsimp}\, c \stackrel{\text{def}}{=} rsimp\ \ (r\backslash_r\, c)$$

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

We do not define an r-regular expression version of *blexer_simp*, as our proof does not depend on it. Now we are ready to introduce how r-regular expressions allow us to prove the size bound on bitcoded regular expressions.

6.2.3 Using R-regular Expressions to Bound Bit-coded Regular Expressions

Everything about the size of annotated regular expressions after the application of function *bsimp* and $\backslash_{simps}$ can be calculated via the size of r-regular expressions after the application of *rsimp* and $\backslash_{rsimps}$:

Lemma 6. *The following equalities hold:*

- $[\![(a)_{\downarrow_r}]\!]_r = [\![a]\!]$
- $[\![bsimp\ a]\!] = [\![rsimp\ (a)_{\downarrow_r}]\!]_r$

- $[\![a\backslash_{bsimps} s]\!] = [\![(a)_{\downarrow_r}\backslash_{rsimps} s]\!]_r$

Proof. First part follows from the definition of $(\_)_{\downarrow_r}$. The second part is by induction on the inductive cases of *bsimp*. The third part is by induction on the string s, where the inductive step follows from part one. □

With lemma 6, we will be able to focus on estimating only $[\![(a)_{\downarrow_r}\backslash_{rsimps} s]\!]_r$ in later parts because

$$[\![(a)_{\downarrow_r}\backslash_{rsimps} s]\!]_r \leq N_r \quad \text{implies} \quad [\![a\backslash_{bsimps} s]\!] \leq N_r.$$

If we attempt to prove

$$\forall r.\ \exists N_r.\ \ s.t. [\![r\backslash_{rsimps} s]\!]_r \leq N_r$$

using a naive induction on the structure of r, then we are stuck at the inductive cases such as $r_1 \cdot r_2$. The inductive hypotheses are:

$$\begin{array}{l}\text{1: for } r_1\text{, there exists } N_{r_1}.\ \ s.t.\ \ \forall s. [\![r_1\backslash_{rsimps} s]\!]_r \leq N_{r_1}.\\ \text{2: for } r_2\text{, there exists } N_{r_2}.\ \ s.t.\ \ \forall s. [\![r_2\backslash_{rsimps} s]\!]_r \leq N_{r_2}.\end{array}$$

The inductive step to prove would be

$$\text{there exists } N_{r_1\cdot r_2}.\ \ s.t. \forall s. [\![(r_1 \cdot r_2)\backslash_{rsimps} s]\!]_r \leq N_{r_1\cdot r_2}.$$

The problem is that it is not clear what $(r_1 \cdot r_2)\backslash_{rsimps} s$ looks like, and therefore N_{r_1} and N_{r_2} in the inductive hypotheses cannot be directly used.

The point however, is that they will be equivalent to a list of terms $\sum rs$, where each term in rs will be made of $r_1\backslash s'$, $r_2\backslash s'$, and $r\backslash s'$ with $s' \in SubString\ s$ (which stands for the set of substrings of s). The list $\sum rs$ will then be de-duplicated by *rdistinct* in the simplification, which prevents the rs from growing indefinitely.

Based on this idea, we develop a proof in two steps. First, we show the below equality (where f and g are functions that do not increase the size of the input)

$$r\backslash_{rsimps} s = f\ (rdistinct\ (g\ \sum rs)),$$

where $r = r_1 \cdot r_2$ or $r = r_0^*$ and so on. For example, for $r_1 \cdot r_2$ we have the equality as

$$r_1 \cdot r_2\backslash_{rsimps} s = rsimp\ (\sum(r_1\backslash s \cdot r_2)\ ::\ (map\ r_2\backslash_{rsimps\_}\ (Suffix\ s\ r_1)))$$

We call the right-hand-side the *Closed Form* of $(r_1 \cdot r_2)\backslash_{rsimps} s$. Second, we will bound the closed form of r-regular expressions using some estimation techniques and then apply lemma 6 to show that the bitcoded regular expressions in our *blexer_simp* are finitely bounded.

We will describe in detail the first step of the proof in the next section.

6.3 Closed Forms

In this section we introduce in detail how to express the string derivatives of regular expressions (i.e. $r\backslash_r s$ where s is a string rather than a single character) in a different way than our previous definition. In previous chapters, the derivative of a regular expression r w.r.t a string s was recursively defined on the string:

$$r\backslash_s(c :: s) \stackrel{\text{def}}{=} (r\backslash c)\backslash_s s$$

The problem is that this definition does not provide much information on what $r\backslash_s s$ looks like. If we are interested in the size of a derivative like $(r_1 \cdot r_2)\backslash s$, we have to somehow get a more concrete form to begin. We call such more concrete representations the "closed forms" of string derivatives as opposed to their original definitions. The name "closed from" was inspired by closed forms in math, and the similarity with closed forms here is that they make estimating the same term easier.

We start by proving some basic identities involving the simplification functions for r-regular expressions. After that we introduce the rewrite relations $\rightsquigarrow_h$, $\rightsquigarrow^*_{scf}$ $\rightsquigarrow_f$ and $\rightsquigarrow_g$. These relations involve similar techniques as in chapter 5 for annotated regular expressions. Finally, we use these identities to establish the closed forms of the alternative regular expression, the sequence regular expression, and the star regular expression.

6.3.1 Some Basic Identities

In what follows we will often convert between lists and sets. We use Isabelle's *set* to refer to the function that converts a list *rs* to the set containing all the elements in *rs*.

rdistinct's Does the Job of De-duplication

The *rdistinct* function, as its name suggests, will return a list of r-regular expressions whose elements are distinct. The function application *rdistinct rs acc* returns a list of distinct regular expressions. The accumulator set *acc* stands for the set of regular expressions "accumulated" so far, and therefore should be removed from *rs*.

We present the properties about *rdistinct* as a list of sub-lemmas. Sub-lemmas 14 and 15 are the most important of this list–they will be used in equalities that we prove about *rsimp*. But we choose to put some of their necessary preliminary lemmas in as they are interesting in their own right. Particularly interesting is sub-lemma 8, as the two propositions in each point need to be proven together to allow the induction to go through. The key takeaway of this lemma is that *rdistinct* does what it is supposed to do–deduplication.

Lemma 7. *Assume we have the predicate isDistinct*[1] *for testing whether a list's elements are unique. Then the following properties about rdistinct hold:*

1. *If* $a \in acc$ *then* $a \notin (rdistinct\ \ rs\ \ acc)$.
2. $isDistinct\ \ (rdistinct\ \ rs\ \ acc)$.
3. $set\ (rdistinct\ \ rs\ \ acc) = (set\ rs) - acc$
4. *The elements appearing in the accumulator will always be removed. More precisely, If* $rs \subseteq rset$, *then* $rdistinct\ \ rs@rsa\ \ acc = rdistinct\ \ rsa\ \ acc$.
5. *More generally, if* $a \in rset$ *and* $rdistinct\ \ rs\ \ \{a\} = []$, *then* $rdistinct\ \ (rs@rs')\ \ rset = rdistinct\ \ rs'\ \ rset$
6. *The accumulator can be augmented to include elements not appearing in the input list, and the output will not change. If* $r \notin rs$, *then* $rdistinct\ \ rs\ \ acc = rdistinct\ \ rs\ \ (\{r\} \cup acc)$.

[1] We omit its recursive definition here. Its Isabelle counterpart would be *distinct*.

7. *Particularly, if isDistinct rs, then we have*

$$rdistinct\ \ rs\ \ \varnothing = rs$$

8. *The two properties hold if $r \in rs$:*

   - $rdistinct\ \ rs\ \ rset = rdistinct\ \ (rs@[r])\ \ rset$
     and
     $rdistinct\ \ (ab :: rs@[ab])\ \ rset' = rdistinct\ \ (ab :: rs)\ \ rset'$
   - $rdistinct\ \ (rs@rs')\ \ rset = rdistinct\ \ rs@[r]@rs'\ \ rset$
     and
     $rdistinct\ \ (ab :: rs@[ab]@rs'')\ \ rset' = rdistinct\ \ (ab :: rs@rs'')\ \ rset'$

9. $rdistinct\ \ (rs@rs')\ \ \varnothing = rdistinct\ \ ((rdistinct\ \ rs\ \ \varnothing)@rs')\ \ \varnothing$

10. $rdistinct\ \ (rs@rs')\ \ \varnothing = rdistinct\ \ (rdistinct\ \ rs\ \ \varnothing@rs')\ \ \varnothing$

11. *If $rset' \subseteq rset$, then $rdistinct\ \ rs\ \ rset = rdistinct\ \ (rdistinct\ \ rs\ \ rset')\ \ rset$. As a corollary of this,*

12. $rdistinct\ \ (rs@rs')\ \ rset = rdistinct\ \ (rdistinct\ \ rs\ \ \varnothing)@rs'\ \ rset$. *This gives another corollary use later:*

13. *If $a \in rset$, then $rdistinct\ \ (rs@rs')\ \ rset = rdistinct\ \ (rdistinct\ \ (a :: rs)\ \ \varnothing@rs')\ \ rset$,*

14. *rdistinct is composable w.r.t list concatenation: If isDistinct rs_1, and $(set\ rs_1) \cap acc = \varnothing$, then applying rdistinct on $rs_1@rs_a$ does not have an effect on rs_1:*

$$rdistinct\ (rs_1@rsa)\ \ acc = rs_1@(rdistinctrsa\ (acc \cup rs_1))$$

15. *rdistinct needs to be applied only once, and applying it multiple times does not make any difference: $rdistinct\ (rs@rs_a)\ \varnothing = rdistinct\ ((rdistinct\ rs\ \varnothing)@(rdistinct\ rs_a\ (set\ rs)))\ \varnothing$*

Proofs of these are omitted, as they can be completed by straightforward induction/automation. This is somewhat similar to the classical theorem prover tutorial example $rev\ (rev\ rs) = rs$: as long as the necessary preliminary lemmas such as $rev(rs@rs') = rev\ rs'@rev\ rs$ that needs to be proven beforehand are outlined, it is routine to flesh out the detailed proofs. In a similar vein we do not provide the flesh of the proofs but give an outline.

The Properties of *Rflts*

We give in this subsection some properties involving $\backslash_r$, $\backslash_{rsimps}$, *rflts* and $rsimp_{ALTS}$, together with any non-trivial lemmas that lead to them. These will be helpful in later closed-form proofs, when we want to transform derivative terms which have interleaving derivatives and simplifications applied to them.

Lemma 8. *The function rflts has the properties below:*

- *Rflts is composable in terms of concatenation: $rflts\ (rs_1@rs_2) = rflts\ rs_1@rflts\ rs_2$*
- *If $r \neq \mathbf{0}_r$ and $\nexists rs_1.r = \sum rs_1$, then $rflts\ (r :: rs) = r :: rflts\ rs$*
- $rflts\ (rs@[\mathbf{0}_r]) = rflts\ rs$

- *rflts* $(rs'@[\sum rs]) = rflts\ rs'@rs$
- *rflts* $(rs@[\mathbf{1}_r]) = rflts\ rs@[\mathbf{1}_r]$
- *If* $r \neq \mathbf{0}_r$ *and* $\nexists rs'.r = \sum rs'$ *then rflts* $(rs@[r]) = (rflts\ rs)@[r]$
- *If* $r = \sum rs$ *and* $r \in rs'$ *then for all* $r_1 \in rs.r_1 \in rflts\ rs'$.
- *rflts* $(rs_a@\mathbf{0}_r :: rs_b) = rflts\ (rs_a@rs_b)$
- *The derivative operation and the* $rsimp_{ALTS}$ *function commute:*

$$(rsimp_{ALTS}\ rs)\backslash_r x = rsimp_{ALTS}\ (map\ (\_\backslash_r x)\ rs)$$

(this will be used later on when deriving the closed form for the alternative regular expression)

Simplified *Rrexp*s are Good

We formalise the notion of "good" regular expressions, which characterise regular expressions that have been simplified once by *rsimp*. The definition of *good* is:

$$\begin{array}{lcl}
good\ \mathbf{0}_r & \stackrel{\text{def}}{=} & false \\
good\ \mathbf{1}_r & \stackrel{\text{def}}{=} & true \\
good\ \mathbf{c}_r & \stackrel{\text{def}}{=} & true \\
good\ \sum[] & \stackrel{\text{def}}{=} & false \\
good\ \sum[r] & \stackrel{\text{def}}{=} & false \\
good\ \sum r_1 :: r_2 :: rs & \stackrel{\text{def}}{=} & isDistinct\ (r_1 :: r_2 :: rs) \\
 & & \wedge\ (\forall r' \in (r_1 :: r_2 :: rs).\ good\ r' \ \wedge\ nonAlt\ r') \\
good\ \mathbf{0}_r \cdot r & \stackrel{\text{def}}{=} & false \\
good\ \mathbf{1}_r \cdot r & \stackrel{\text{def}}{=} & false \\
good\ r \cdot \mathbf{0}_r & \stackrel{\text{def}}{=} & false \\
good\ r_1 \cdot r_2 & \stackrel{\text{def}}{=} & good\ r_1\ and\ good\ r_2 \\
good\ r^* & \stackrel{\text{def}}{=} & true
\end{array}$$

For alternative regular expressions that means they do not contain any nested alternatives, un-eliminated $\mathbf{0}_r$s, singleton or empty children list, or duplicate elements (for example, $r_1 + (r_2 + r_3)$, $\mathbf{0}_r + r$, $\sum[]$ or $\sum[r, r, \ldots]$). We omit the recursive definition of the predicate *nonAlt*, which evaluates to true when the regular expression is not an alternative, and false otherwise.

We prove that all simplified r-regular expressions are *good* unless they are $\mathbf{0}_r$. In other words, *good* precisely characterises the features of *rsimp*'s output:

Lemma 9. *(Simplified regular expressions are good.)*
For any r-regular expression r, good rsimp r or rsimp $r = \mathbf{0}_r$.

Proof. By an induction on r. For the sequence case,

$$rsimp\ r_1 \cdot r_2 = rsimp_{SEQ}\ rsimp\ r_1\ rsimp\ r_2.$$

If at least one of *rsimp* r_1 or *rsimp* r_2 is $\mathbf{0}_r$ then RHS is $\mathbf{0}_r$. Otherwise both are *good* and therefore *rsimp* $r_1 \cdot r_2 = RSEQ\ rsimp\ r_1\ rsimp\ r_2$ is *good*.

For the alternative case, $rsimp\ r = rsimp\ _{ALTS}\ (rdistinct\ \ rflts\ (map\ rsimp\ \ rs)\ \ \varnothing)$. Let $rs' = map\ rsimp\ \ rs$. By inductive hypothesis, all r' from rs' are *good* or $\mathbf{0}_r$s. Let $rs'' = rflts\ rs'$. By definition of *rflts* and *good*, all elements from rs'' are *good* and not $\mathbf{0}_r$s or alternatives. Let $rs''' = rdistinct\ \ rs''\ \ \varnothing$, then all elements in rs''' are distinct by lemma 7, and they are *good*. Now we do a case analysis on rs'''. It could be an empty list, which means $rsimp\ r = \mathbf{0}_r$, or a singleton list r_s, which means $rsimp\ r = r_s$ is *good*. Or it could be a list with two or more elements, and $rsimp\ r = \sum rs'''$ which is *good* by definition. □

Now we are ready to prove that good regular expressions are a fixed point for the function *rsimp* , and more importantly *rsimp* is idempotent:

Lemma 10. *(Good r-regular expressions cannot be further simplified.)*

- *If good r then rsimp r = r.*
- *(rsimp idempotency) rsimp (rsimp r) = rsimp r*

Proof. For the first part, By an induction on the inductive cases of *good*. The most involved case is for the alternative. Let $r = \sum rs$ be a *good* alternative. we have the inductive hypothesis that $\forall r \in rs.good\ r$, and additionally: (i)rs is not an empty or singleton list, (ii)all elements of rs are not alternative regular expressions or $\mathbf{0}_r$s, (iii)all regular expressions in rs are distinct, (iv)$map\ rsimp\ \ rs = rs$. By (ii) we know that $rflts\ rs = rs$, and by (iii) we have $rdistinct\ \ rflts\ rs\ \ \varnothing = rs$.

Therefore

$$\begin{array}{ll}
rsimp\ r & = \\
rsimp\ \sum rs & = \\
rsimp\ _{ALTS}\ (rdistinct\ \ rflts\ (map\ rsimp\ \ rs)\ \ \varnothing) & = \\
rsimp\ _{ALTS}\ (rdistinct\ \ rflts\ rs\ \ \varnothing) & = \\
rsimp\ _{ALTS}\ (rdistinct\ \ rs\ \ \varnothing) & = \\
rsimp\ _{ALTS}\ rs & = \\
\sum rs. &
\end{array}$$

Part 2 follows from part 1 because we know that $r' = rsimp\ r$ is either *good* or $\mathbf{0}_r$ from lemma 9. In both cases, we have $rsimp\ r' = r'$. □

The idempotency of *rsimp* is very useful in proving equality for closed forms. For instance, one can use the equality $rsimp\ rsimp\ r = rsimp\ r$ in a symmetric way, and obtain properties like:

Corollary 2.

- $map\ rsimp\ (r :: rs) = map\ rsimp\ \ (rsimp\ r :: rs)$
- $rsimp\ (\sum rs) = rsimp\ (\sum map\ rsimp\ \ rs)$.
- $rsimp_{ALTS}\ (\mathbf{0}_r :: rs) \overset{rsimp}{=} rsimp_{ALTS}\ rs$
- $rsimp_{ALTS}\ rs \overset{rsimp}{=} rsimp_{ALTS}(map\ rsimp\ \ rs)$
- $\sum\sum rs \overset{rsimp}{=} \sum rs$
- $\sum((\sum rs_a) :: rs_b) \overset{rsimp}{=} \sum rs_a @ rs_b$

- $\sum rs \overset{rsimp}{=} \sum map\ rsimp\ \ rs$

The idempotency property also means the fixed-point construction in Sulzmann and Lu's simplification function is strictly unnecessary for *bsimp*, as one does not get any more gains with multiple consecutive simplifications.

The idempotency lemma also allows us to prove some of the most involved properties that need to be used in the closed-form proofs. The idea is that the way alternatives are nested do not matter under *rsimp* :

$$(a + b) + (c + d) \overset{\text{rsimp}}{=} \sum[a, b, c, d].$$

This is very intuitive, however surprisingly hard to prove in our formalisation. In fact, we found this below lemma the most complicated to prove in the entire formalisation of this thesis:

Lemma 11. *One can flatten the inside $\sum$ of a $\sum$ if it is being simplified. Concretely,*

- *If for all $r \in rs, rs', rs''$, we have good r or $r = \mathbf{0}_r$, then $\sum(rs'@rs@rs'') \overset{rsimp}{=} \sum(rs'@[\sum rs]@rs'')$ holds. As a corollary,*
- $\sum(rs'@[\sum rs]@rs'') \overset{rsimp}{=} \sum(rs'@rs@rs'')$

Proof. By rewriting steps involving the use of 10 and 9.

$$\begin{aligned}
&rsimp\ \sum(rs'@rs@rs'') && = \\
&rsimp_{ALTS}\ (rdistinct\ \ rflts\ ((map\ rsimp\ \ rs')@(map\ rsimp\ \ rs)@(map\ rsimp\ \ rs''))\ \varnothing) && = \\
&rsimp_{ALTS}\ (rdistinct\ \ (rflts\ rs'@rflts\ rs@rflts\ rs'')\ \ \varnothing) && = \\
&rsimp_{ALTS}\ (rdistinct\ \ (rflts\ rs'@rdistinct\ \ (rflts\ rs)\ \ \varnothing@rflts\ rs'')\ \ \varnothing) && = \\
&rsimp_{ALTS}\ (rdistinct\ \ (rflts\ rs'@rdistinct\ \ (rflts\ (map\ rsimp\ \ rs))\ \ \varnothing@rflts\ rs'')\ \ \varnothing) && = \\
&rsimp_{ALTS}\ (rdistinct\ \ (rflts\ rs'@[rsimp_{ALTS}\ (rdistinct\ \ (rflts\ (map\ rsimp\ \ rs))\ \ \varnothing)]@rflts\ rs'')\ \ \varnothing) && = \\
&rsimp_{ALTS}\ (rdistinct\ \ (rflts\ rs'@[rsimp\ \sum rs]@rflts\ rs'')\ \ \varnothing) && = \\
&rsimp_{ALTS}\ (rdistinct\ \ (rflts\ (rs'@[\sum rs]@rs''))\ \ \varnothing) && = \\
&rsimp\ \sum(rs'@[\sum rs]@rs'')
\end{aligned}$$

rsimp can be added and removed freely because *rs*'s elements are *good* (or $\mathbf{0}_r$), so *map rsimp rs* = *rs*. Getting part 2 from part 1 is by

$$rsimp\ \sum(rs'@rs@rs'') = rsimp\ \sum(rsimp\ rs'@rsimp\ rs@rsimp\ rs'').$$

Then we know that all elements of *rsimp rs* etc. are *good* and therefore part 1 applies. □

It seems to us that the reason for the hardness was this lemma cannot be broken down into several smaller properties. Therefore the term *rsimp* $(\sum rs'@rs@rs'')$ has to be expanded manually to guide the proof or otherwise the sheer size of the expanded terms and the different possibilities of expanding would cause sledgehammer to time out. Even though the steps in the proof of this lemma is reasonably detailed, it is still not a verbatim representation of the rewrite steps in the Isabelle

RSEQ0L
$\mathbf{0}_r \cdot r_2 \rightsquigarrow_h \mathbf{0}_r$

RSEQ0R
$r_1 \cdot \mathbf{0}_r \rightsquigarrow_h \mathbf{0}_r$

RSEQ1
$(\mathbf{1}_r \cdot r) \rightsquigarrow_h r$

RSEQL
$$\frac{r_1 \rightsquigarrow_h r_2}{r_1 \cdot r_3 \rightsquigarrow_h r_2 \cdot r_3}$$

RSEQR
$$\frac{r_3 \rightsquigarrow_h r_4}{r_1 \cdot r_3 \rightsquigarrow_h r_1 \cdot r_4}$$

RALTSCHILD
$$\frac{r \rightsquigarrow_h r'}{\sum(rs_1 @ [r] @ rs_2) \rightsquigarrow_h \sum(rs_1 @ [r'] @ rs_2)}$$

RALTS0
$\sum(rs_a @ [\mathbf{0}_r] @ rs_b) \rightsquigarrow_h \sum(rs_a @ rs_b)$

RALTSNESTED
$\sum(rs_a @ [\sum rs_1] @ rs_b) \rightsquigarrow_h \sum(rs_a @ rs_1 @ rs_b)$

RALTSNIL
$\sum[] \rightsquigarrow_h \mathbf{0}_r$

RALTSSINGLE
$\sum[r] \rightsquigarrow_h r$

RALTSDELETE
$$\frac{r_1 = r_2}{\sum rs_a @ [r_1] @ rs_b @ [r_2] @ rsc \rightsquigarrow_h \sum rs_a @ [r_1] @ rs_b @ rs_c}$$

FIGURE 6.2: List of one-step rewrite rules for r-regular expressions ($\rightsquigarrow_h$)

proof script. Typesetting all detailed steps here is too cumbersome. We refer the interested readers to the lemma called "good_flatten_middle" in the repository [46] for details.

We need more equalities like the above lemma to enable a closed form lemma, for which we need to introduce a few rewrite relations to help us obtain them.

6.3.2 The rewrite relation $\rightsquigarrow_h$, $\overset{*}{\rightsquigarrow}_{scf}$, $\rightsquigarrow_f$ and $\rightsquigarrow_g$

Inspired by the success we had in the correctness proof of *blexer_simp* in chapter 5, we follow suit here, defining individual simplification steps as "small-step" rewriting rules. This allows capturing similarities between terms that would be otherwise hard to express.

We use $\rightsquigarrow_h$ for one-step rewrite of regular expression simplification, $\rightsquigarrow_f$ for rewrite of list of regular expressions that include all operations carried out in *rflts*, and $\rightsquigarrow_g$ for rewriting a list of regular expressions possible in both *rflts* and *rdistinct*. Their reflexive transitive closures are used to denote zero or many steps, as was the case in the previous chapter. As we have already done something similar, the presentation about these rewriting rules will be more concise than that in 5. To differentiate between the rewriting steps for annotated regular expressions and *rrexp*s, we add characters h and g below the squig arrow symbol to mean simplification transitions of *rrexp*s and *rrexp* lists, respectively.

Like $\rightsquigarrow_s$, it is convenient to define rewrite rules for a list of regular expressions, where each element can rewrite in many steps to the other (scf stands for list *c*losed *f*orm). This relation is similar to the $\overset{s*}{\rightsquigarrow}$ for annotated regular expressions.

List of one-step rewrite rules for flattening a list of regular expressions($\rightsquigarrow_f$):

$$[] \rightsquigarrow^*_{scf} [] \qquad \frac{r \rightsquigarrow^*_h r' \qquad rs \rightsquigarrow^*_{scf} rs'}{r :: rs \rightsquigarrow^*_{scf} r' :: rs'}$$

FIGURE 6.3: List of one-step rewrite rules for a list of r-regular expressions

$$\mathbf{0}_r :: rs \rightsquigarrow_f rs \qquad (\sum rs) :: rs_a \rightsquigarrow_f rs@rs_a \qquad \frac{rs_1 \rightsquigarrow_f rs_2}{r :: rs_1 \rightsquigarrow_f r :: rs_2}$$

FIGURE 6.4: List of one-step rewrite rules characterising the *rflts* operation on a list

Lists of one-step rewrite rules for flattening and de-duplicating a list of regular expressions ($\rightsquigarrow_g$):

$$\mathbf{0}_r :: rs \rightsquigarrow_g rs \qquad (\sum rs) :: rs_a \rightsquigarrow_g rs@rs_a \qquad \frac{rs_1 \rightsquigarrow_g rs_2}{r :: rs_1 \rightsquigarrow_g r :: rs_2}$$

$$\text{DB}\quad rs_a@[a]@rs_b@[a]@rs_c \rightsquigarrow_g rs_a@[a]@rsb@rsc$$

FIGURE 6.5: List of one-step rewrite rules characterising the *rflts* and *rdistinct* operations

We define two separate list rewriting relations $\rightsquigarrow_f$ and $\rightsquigarrow_g$. The rewriting steps that take place during flattening are characterised by $\rightsquigarrow_f$. The rewrite relation $\rightsquigarrow_g$ characterises both flattening and de-duplicating. Sometimes $\rightsquigarrow^*_g$ is slightly too powerful so we would rather use $\rightsquigarrow^*_f$ to prove equalities related to *rflts*. For instance, $\rightsquigarrow_f$ is more useful in proving the below lemma:

Lemma 12. $\sum(rdistinct\ \ (map\ (\_\backslash x)\ (rflts\ rs))\ \ \varnothing) \overset{rsimp}{=} \sum(rdistinct\ \ (rflts\ (map\ (\_\backslash x)\ rs))\ \ \varnothing)$

Proof. We have that

$$map\ (\_\backslash x)\ (rflts\ rs) \rightsquigarrow^*_f rflts\ (map\ (\_\backslash x)\ rs)$$

holds. Also, we have that

$$rs_1 \rightsquigarrow^*_f rs_2 \implies \sum(rdistinct\ rs_1\ \varnothing) \overset{\text{rsimp}}{=} \sum(rdistinct\ rs_2\ \varnothing).$$

By letting $rs_1 = (rdistinct\ \ (map\ (\_\backslash x)\ (rflts\ rs))\ \ \varnothing)$ and $rs_2 = (rdistinct\ \ (rflts\ (map\ (\_\backslash x)\ rs))\ \ \varnothing)$, we get the desired equality. □

But this trick will not work for $\rightsquigarrow^*_g$. For example, a rewriting step in proving closed forms is:

$$rsimp\ (rsimp_{ALTS}\ (map\ (\_\backslash x)\ (rdistinct\ \ (rflts\ (map\ (rsimp\ \ \circ\ (\lambda r.r\backslash_{rsimps} xs))))\ \varnothing)))$$
$$=$$
$$rsimp\ (rsimp_{ALTS}\ (rdistinct\ \ (map\ (\_\backslash x)\ (rflts\ (map\ (rsimp\ \ \circ\ (\lambda r.r\backslash_{rsimps} xs)))))\ \varnothing))$$

For this, one would hope to have a rewriting relation between the two lists involved, similar to 6.3.2. However, it turns out that

$$map\ (\_\backslash x)\ (rdistinct\ rs\ rset) \rightsquigarrow_g^* rdistinct\ (map\ (\_\backslash x)\ rs)\ (rset\backslash x)$$

does **not** hold in general. For this rewriting step we will introduce some slightly more cumbersome proof technique later. The point is that $\rightsquigarrow_f$ allows us to prove equivalence in a straightforward way that is not possible for $\rightsquigarrow_g$.

Terms That Can Be Rewritten Using $\rightsquigarrow_h^*$, $\rightsquigarrow_g^*$, and $\rightsquigarrow_f^*$

In this part, we present lemmas stating pairs of r-regular expressions and r-regular expression lists where one can rewrite from one in many steps to the other. Most of the proofs to these lemmas are straightforward, using an induction on the corresponding rewriting relations. These proofs will therefore be omitted when this is the case. Also because we have shown similar techniques in chapter 5 with rewriting relations, the presentation of proofs will be accelerated and more concise. Below are some properties of $\rightsquigarrow_g^*$, $\rightsquigarrow_g^*$, $\rightsquigarrow_{scf}^*$, and $\rightsquigarrow_f^*$:

Lemma 13.

- $rs_1 @ rs \rightsquigarrow_g^* rs_1 @ (rdistinct\ rs\ rs_1)$
- $rs \rightsquigarrow_g^* rdistinct\ rs\ \varnothing$
- $rs_a @ (rdistinct\ rs\ rs_a) \rightsquigarrow_g^* rs_a @ (rdistinct\ rs\ (\{\mathbf{0}_r\} \cup rs_a))$
- $rs\ @\ rdistinct\ rs_a\ rset \rightsquigarrow_g^* rs @ rdistinct\ rs_a\ (rest \cup rs)$
- *If a pair of terms rs_1, rs_2 are rewritable via $\rightsquigarrow_g^*$ to each other, then they are equivalent under rsimp : If $rs_1 \rightsquigarrow_g^* rs_2$, then we have the following equivalence:*
  - $\sum rs_1 \overset{rsimp}{=} \sum rs_2$
  - $rsimp_{ALTS}\ rs_1 \overset{rsimp}{=} rsimp_{ALTS}\ rs_2$

  Here are a few connecting lemmas showing that if a list of regular expressions can be rewritten using $\rightsquigarrow_g^*$ or $\rightsquigarrow_f^*$ or $\rightsquigarrow_{scf}^*$, then an alternative constructor taking the list can also be rewritten using $\rightsquigarrow_h^*$:
  - *If $rs \rightsquigarrow_g^* rs'$ then $\sum rs \rightsquigarrow_h^* \sum rs'$.*
  - *If $rs \rightsquigarrow_g^* rs'$ then $\sum rs \rightsquigarrow_h^* rsimp_{ALTS}\ rs'$*
  - *If $rs_1 \rightsquigarrow_{scf}^* rs_2$ then $\sum (rs @ rs_1) \rightsquigarrow_h^* \sum (rs @ rs_2)$*
  - *If $rs_1 \rightsquigarrow_{scf}^* rs_2$ then $\sum rs_1 \rightsquigarrow_h^* \sum rs_2$*
- *If $r_1 \rightsquigarrow_h^* r_2$ then $r_1 \overset{rsimp}{=} r_2$.*
- *Similar to chapter 5, we prove that r-derivatives and rewriting commute:*
  If $r \rightsquigarrow_h^* r'$ then $r\backslash_r c \rightsquigarrow_h^* r'\backslash_r c$.

- $rsimp\ ((rsimp\ r)\backslash_r c) = rsimp\ (r\backslash_r c)$
- *As corollaries of the above lemma, we have*
  - *If* $s \neq []$ *then* $r\backslash_{rsimps} s = rsimp\ (r\backslash_{rsimps} s)$.
  - $rsimp_{ALTS}\ (map\ (\_\backslash_r x)\ (rdistinct\ \ rs\ \ \varnothing)) \overset{rsimp}{=} rsimp_{ALTS}\ (rdistinct\ (map\ (\_\backslash_r x)\ rs)\ \varnothing)$

Generally these lemmas are simple enough that they can be solved by straightforward induction and automation. They will be used at later proofs as individual rewriting steps.

6.3.3 Closed Forms for $\sum rs$, $r_1 \cdot r_2$ and r^*

Lemma 13 leads to our first closed form, which is for the alternative regular expression:

Theorem 5.

$$(\sum rs)\backslash_{rsimps} s \overset{rsimp}{=} \sum\ (map\ (\_\backslash_{rsimps} s)\ rs)$$

Proof. By an induction on the string s. The case split is [] ans $s@[c]$ (this induction strategy is supported by Isabelle's list type). For the base case,

$$LHS = ((\sum rs)\backslash_{rsimps} []) = (\sum rs) = \sum\ (map\ id\ rs) = RHS.$$

The inductive case is proven by the following rewrite stpes:

$$\begin{array}{ll}
LHS & = \\
\sum rs\backslash_{rsimps}(s@[c]) & = \\
(\sum rs\backslash_{rsimps} s)\backslash_{rsimps}[c] & = \\
(\sum(map\ (_\backslash_{rsimps} s)\ rs))\backslash_{rsimps}[c] & = \\
rsimp\ \ ((\sum(map\ (_\backslash_{rsimps} s)\ rs))\backslash c) & = \\
rsimp_{ALTS}\ (map\ (_\backslash c)\ (rdistinct\ \ (rflts\ (map\ (rsimp\ \ \circ\ (\lambda r.r\backslash_{rsimps} s))))\ \ \varnothing)) & = \\
rsimp_{ALTS}\ (rdistinct\ \ (map\ (_\backslash c)\ (rflts\ (map\ (rsimp\ \ \circ\ (\lambda r.r\backslash_{rsimps} s))\ rs)))\ \ \varnothing) & = \\
rsimp_{ALTS}\ (rdistinct\ \ (rflts\ (map\ ((_\backslash c) \circ rsimp\ \circ (\lambda r.r\backslash_{rsimps} s))\ rs))\ \ \varnothing) & = \\
rsimp_{ALTS}\ (rdistinct\ \ (rflts\ (map\ (rsimp\ \circ (_\backslash c) \circ (\lambda r.r\backslash_{rsimps} s))\ rs))\ \ \varnothing) & = \\
rsimp_{ALTS}\ (rdistinct\ \ (rflts\ (map\ ((\lambda r.r\backslash_{rsimps}(s@[c])))\ rs))\ \ \varnothing) & = \\
rsimp\ \ (map\ ((\lambda r.r\backslash_{rsimps}(s@[c])))\ rs). &
\end{array}$$

□

This closed form has a variant which can be more convenient in later proofs:

Corollary 3. *If* $s \neq []$, *then* $(\sum\ rs)\backslash_{rsimps} s = rsimp\ (\sum\ (map\ \_\backslash_{rsimps} s\ rs))$.

Here are some examples of this closed form:

$$\begin{array}{lll}
(a^* + b^*)\backslash_{rsimps} aaaa & \overset{\text{rsimp}}{=} & (a^*\backslash_{rsimps} aaaa + b^*\backslash_{rsimps} aaaa) \\
(aba + ab + a)\backslash_{rsimps} aba & \overset{\text{rsimp}}{=} & (aba\backslash_{rsimps} aba + ab\backslash_{rsimps} aba + a\backslash_{rsimps} aba)
\end{array}$$

Despite the intuition being quite simple, the precise formulation and proof is rather heavy. The harder closed forms are the sequence and star ones. Before we obtain them, some preliminary definitions are needed to make proof statements concise.

Closed Form for Sequence Regular Expressions

For the sequence regular expression, let's first look at a series of derivative steps on it (assuming that each time when a derivative is taken, the head of the sequence is always nullable):

$$
\begin{array}{ll}
r_1 \cdot r_2 & \longrightarrow_{\backslash c} \\
r_1\backslash c \cdot r_2 + r_2\backslash c & \longrightarrow_{\backslash c'} \\
(r_1\backslash cc' \cdot r_2 + r_2\backslash c') + r_2\backslash cc' & \longrightarrow_{\backslash c''} \\
((r_1\backslash cc'c'' \cdot r_2 + r_2\backslash c'') + r_2\backslash c'c'') + r_2\backslash cc'c'' & \longrightarrow_{\backslash c''} \\
\ldots &
\end{array}
$$

Roughly speaking $r_1 \cdot r_2\backslash s$ can be expressed as a giant alternative taking a list of terms $[r_1\backslash_r s \cdot r_2, r_2\backslash_r s'', r_2\backslash_r s_1'', \ldots]$, where the head of the list is always the term representing a match involving only r_1, and the tail of the list consisting of terms of the shape $r_2\backslash_r s''$, s'' being a suffix of s. This intuition is also echoed by Murugesan and Sundaram [61], where they gave a pencil-and-paper derivation of $(r_1 \cdot r_2)\backslash s$:

$$
\begin{array}{ll}
L\ [(r_1 \cdot r_2)\backslash_r (c_1 :: c_2 :: \ldots c_n)] & = \\
 & \\
L\ [((r_1\backslash_r c_1) \cdot r_2 + (\delta\ (nullable\ r_1)\ (r_2\backslash_r c_1)))\backslash_r (c_2 :: \ldots c_n)] & = \\
L\ [((r_1\backslash_r c_1c_2 \cdot r_2 + (\delta\ (nullable\ r_1)\ (r_2\backslash_r c_1c_2))) & \\
+(\delta\ (nullable\ r_1\backslash_r c)\ (r_2\backslash_r c_2)))\backslash_r (c_3 \ldots c_n)] & \ldots
\end{array}
$$

The δ function returns r when the boolean condition b evaluates to true and $\mathbf{0}_r$ otherwise:

$$
\begin{array}{rcll}
\delta\ b\ r & \stackrel{\text{def}}{=} & r & \textit{if } b \textit{ is true} \\
 & \stackrel{\text{def}}{=} & \mathbf{0}_r & \textit{otherwise}
\end{array}
$$

We are aware that this delta function notation is unconventional, however we keep this notation to be faithful to the original authors' derivation. Expressed as case statements, the third term's argument would look like

$$
((r_1 \cdot r_2)\backslash_r c_1c_2)\backslash_r = \begin{cases}
\begin{cases}
(r_1\backslash c_1c_2) \cdot r_2 + r_2\backslash c_2 + r_2\backslash c_1c_2 & \text{if } nullable\ r_1 \text{ and } nullable\ r_1\backslash c_1 \\
(r_1\backslash c_1c_2) \cdot r_2 + r_2\backslash c_1c_2 & \text{if } nullable\ r_1 \text{ and } \neg(nullable\ r_1\backslash c_1)
\end{cases} \\
\begin{cases}
(r_1\backslash c_1c_2) \cdot r_2 + r_2\backslash c_2 & \text{if } \neg(nullable\ r_1) \text{ and } nullable\ r_1\backslash c_1 \\
(r_1\backslash c_1c_2) \cdot r_2 & \text{if } \neg(nullable\ r_1) \text{ and } \neg(nullable\ r_1\backslash c_1)
\end{cases}
\end{cases}
$$

The case statements are not possible to eliminate if one wants to be precise here and not use the delta function style notation. For instance, an attempt we made was letting the string derivatives be defined in a "convolution" style, for example for the two-character case as

$$(r_1 \cdot r_2)\backslash c_1c_2 = (r_1\backslash c_1c_2 \cdot r_2 + r_1\backslash c_1 \cdot r_2\backslash c_2) + r_1 \cdot r_2\backslash c_1c_2.$$

This is clearly wrong, and the redundant sub-terms should not created in the first place. For closed forms only terms $r_2\backslash_r s''$ satisfying the property

$$\exists s'.\text{such that } s'@s'' = s \text{ and } r_1\backslash s' \text{ is } nullable.$$

should be generated. Given the arguments s and r_1, we can define a recursive function which generates such terms in the right order: [2]

$$\begin{array}{lcl} Suffix & :: & "string \Rightarrow rrexp \Rightarrow string\ list" \\ Suffix\ []\ \_ & = & [] \\ Suffix\ (c :: cs)\ r_1 & = & if\ (rnullable\ r_1)\ then\ (Suffix\ cs\ (r_1\backslash_r c))@[(c :: cs)] \\ & & else\ (Suffix\ cs\ (r_1\backslash_r c)) \end{array}$$

The list starts with shorter suffixes and ends with longer ones, which means the string elements s'' in the list $Suffix\ \ s\ \ r_1$ are sorted in the same order as that of the terms $r_2\backslash s''$ appearing in $(r_1 \cdot r_2)\backslash s$. In essence, $Suffix\ \ \_\ \ \_$ is doing a "virtual derivative" of $r_1 \cdot r_2$, but instead of producing the entire result $(r_1 \cdot r_2)\backslash s$, it only stores strings, with each string s'' representing a term such that $r_2\backslash s''$ is occurring in $(r_1 \cdot r_2)\backslash s$. Here are a few examples of $Suffix\ s\ r$'s computations: Example 1:

$$\begin{array}{ll} Suffix\ aaa\ a^* & = \\ Suffix\ aa\ (\mathbf{1} \cdot a^*)@[(aaa)] & = \\ (Suffix\ a\ (\mathbf{0} \cdot a^* + \mathbf{1} \cdot a^*)@[(aa)])@[(aaa)] & = \\ (Suffix\ []\ (\mathbf{0} \cdot a^* + \mathbf{0} \cdot a^* + \mathbf{1} \cdot a^*))@[a]@[aa]@[aaa] & = \\ [a, aa, aaa]. & \end{array}$$

We have not optimised $Suffix$ since it is only designed for proofs. However the derivatives can grow into cumbersome arguments soon with larger regular expressions, and therefore in the next examples we simplify things in each step. This is not completely rigorous but makes the presentation much clearer. Example 2:

$$\begin{array}{ll} Suffix\ aaa\ (aa)^* & = \\ Suffix\ aa\ (a \cdot (aa)^*)@[(aaa)] & = \\ (Suffix\ a\ ((aa)^*))@[(aaa)] & = \\ (Suffix\ []\ (a \cdot (aa)^*))@[a]@[aaa] & = \\ [a, aaa]. & \end{array}$$

Example 3:

[2] Perhaps a better name for it would be "NullablePrefixSuffix" to differentiate with the list of *all* prefixes of s, but that is a bit too long for a function name and we are yet to find a more concise and easy-to-understand name.

$$
\begin{array}{ll}
\mathit{Suffix}\ \mathit{ababa}\ (\mathit{aba} + \mathit{ab} + \mathit{a})^* & = \\
\mathit{Suffix}\ \mathit{baba}\ ((\mathit{ba} + \mathit{b} + \mathbf{1}) \cdot (\mathit{aba} + \mathit{ab} + \mathit{a})^*)@[\mathit{ababa}] & = \\
\mathit{Suffix}\ \mathit{aba}\ ((\mathit{a} + \mathbf{1}) \cdot (\mathit{aba} + \mathit{ab} + \mathit{a})^*)@[\mathit{baba}]@[\mathit{ababa}] & = \\
\mathit{Suffix}\ \mathit{ba}\ ((\mathit{aba} + \mathit{ab} + \mathit{a})^* + (\mathit{ba} + \mathit{b} + \mathbf{1}) \cdot (\mathit{aba} + \mathit{ab} + \mathit{a})^*)@[\mathit{aba}]@[\mathit{baba}]@[\mathit{ababa}] & = \\
\mathit{Suffix}\ \mathit{a}\ ((\mathit{aba} + \mathit{ab} + \mathit{a})^* + (\mathit{ba} + \mathit{b} + \mathbf{1}) \cdot (\mathit{aba} + \mathit{ab} + \mathit{a})^*)@[\mathit{ba}]@[\mathit{aba}]@[\mathit{baba}]@[\mathit{ababa}] & = \\
\mathit{Suffix}\ []\ ((\mathit{ba} + \mathit{b} + \mathbf{1}) \cdot (\mathit{aba} + \mathit{ab} + \mathit{a})^*)@[\mathit{a}]@[\mathit{ba}]@[\mathit{aba}]@[\mathit{baba}]@[\mathit{ababa}] & = \\
[\mathit{a}, \mathit{ba}, \mathit{aba}, \mathit{baba}, \mathit{ababa}]. &
\end{array}
$$

We can see more clearly from example 3 that indeed we are getting the suffixes, not substrings or prefixes. *Suffix* enables us to obtain the closed-form of sequence regular expressions:

Theorem 6. $(r_1 \cdot r_2)\backslash_{rsimp s} = rsimp\ (\sum((r_1\backslash s) \cdot r_2) :: (map\ (r_2\backslash\_)(Suffix\ s\ r_1)))$

Proof. Let us first define a function $(\!|\ |\!)$ (together with its helper $(\!|\ |\!)'$) which opens up nested alternatives and turn them into a single-level alternative:

$$
\begin{array}{lcl}
(\!|\_|\!)' & :: & "rrexp \Rightarrow rrexp\ list" \\
(\!|\sum r :: rs|\!)' & \stackrel{\text{def}}{=} & (\!|r|\!)'@rs \\
(\!|\sum []|\!)' & \stackrel{\text{def}}{=} & [] \\
(\!|r|\!)' & \stackrel{\text{def}}{=} & [r] \\
(\!|\_|\!) & :: & "rrexp \Rightarrow rrexp" \\
(\!|(\sum r :: rs)|\!) & \stackrel{\text{def}}{=} & \sum((\!|r|\!)'@rs) \\
(\!|\sum []|\!) & \stackrel{\text{def}}{=} & \sum [] \\
(\!|r|\!) & \stackrel{\text{def}}{=} & r
\end{array}
$$

Let $r = (r_1 \cdot r_2)\backslash_r s$ and $rs = (\!|r|\!)'$, then we have that

$$rsimp\ (r_1 \cdot r_2)\backslash_r s = rsimp\ (\sum\ rs)$$

holds. we also have that

$$rs = (r_1\backslash_r s) \cdot r_2 :: (map\ (r_2\backslash_r\_)\ (Suffix\ s\ r_1))$$

holds (see lemma 14). This concludes the proof. □

We present the proof of equation $rs = (r_1\backslash_r s) \cdot r_2 :: (map\ (r_2\backslash_r\_)\ (Suffix\ s\ r_1))$ now, which is required by theorem 6.

Lemma 14. $(\!|(r_1 \cdot r_2)\backslash_r s|\!)' = (r_1\backslash_r s) \cdot r_2 :: (map\ (r_2\backslash_r\_)\ (Suffix\ s\ r_1))$

Proof. The base case holds because

$$(\!|r_1 \cdot r_2|\!)'\backslash_r [] = [r_1 \cdot r_2] = r_1\backslash_s [] \cdot r_2 :: [] = r_1\backslash_s [] \cdot r_2 :: (map\ (r_2\backslash_r\_)\ (Suffix\ []\ r_1)).$$

Now for the inductive case, first define a version of the flattening function that works on a list of r-regular expressions:

$$
\begin{array}{lcl}
(\!|\_|\!)'' & :: & "rrexp\ list \Rightarrow rrexp\ list" \\
(\!|[]|\!)'' & \stackrel{\text{def}}{=} & [] \\
(\!|(r_1 + r_2) :: rs|\!)'' & \stackrel{\text{def}}{=} & r_1 :: r_2 :: rs \\
(\!|r :: rs|\!)'' & \stackrel{\text{def}}{=} & r :: rs
\end{array}
$$

When $r_1 \backslash_r s$ is not *nullable*, we have

$$\begin{array}{ll}
rs & = \\
(\!|(r_1 \cdot r_2)\backslash_r s @ [c]|\!)' & = \\
(\!|((r_1 \cdot r_2)\backslash_r s)\backslash_r c|\!)' & = \\
(\!|(map\ (_\backslash_r c)\ (\!|(r_1 \cdot r_2)\backslash_r s|\!)')|\!)'' & = \\
(\!|(map\ (_\backslash_r c)\ ((r_1\backslash_r s) \cdot r_2 :: (map\ (r_2\backslash_r _)\ (Suffix\ s\ r_1))))|\!)'' & = \\
(\!|(((r_1\backslash_r s @ [c]) \cdot r_2 :: (map\ (r_2\backslash_r _)\ (map\ (\lambda s'.s'@[c])\ (Suffix\ s\ r_1)))))|\!)'' & = \\
(\!|(((r_1\backslash_r s @ [c]) \cdot r_2 :: (map\ (r_2\backslash_r _)\ (Suffix\ s@[c]\ r_1))))|\!)'' & = \\
(r_1\backslash_r s @ [c]) \cdot r_2 :: (map\ (r_2\backslash_r _)\ (Suffix\ s@[c]\ r_1)). &
\end{array}$$

When $r_1 \backslash_r s$ is *nullable*, the antepenultimate step above needs to be changed:

$$\begin{array}{ll}
rs & = \\
(\!|(r_1 \cdot r_2)\backslash_r s @ [c]|\!)' & = \\
\ldots & = \\
(\!|(map\ (_\backslash_r c)\ ((r_1\backslash_r s) \cdot r_2 :: (map\ (r_2\backslash_r _)\ (Suffix\ s\ r_1))))|\!)'' & = \\
(\!|(((r_1\backslash_r s @ [c]) \cdot r_2 :: r_2\backslash_r c :: (map\ (r_2\backslash_r _)\ (map\ (\lambda s'.s'@[c])\ (Suffix\ s\ r_1)))))|\!)'' & = \\
(\!|(((r_1\backslash_r s @ [c]) \cdot r_2 :: (map\ (r_2\backslash_r _)\ (Suffix\ s@[c]\ r_1))))|\!)'' & = \\
(r_1\backslash_r s @ [c]) \cdot r_2 :: (map\ (r_2\backslash_r _)\ (Suffix\ s@[c]\ r_1)). &
\end{array}$$

□

We did not put it before theorem 6 because we wanted to highlight the reason for defining the *Suffix* function–for the closed form of sequence regular expressions. Inserting lemma 14 between the definition of *Suffix* and theorem 6 would make this less clear.

Closed Forms for Star Regular Expressions

The closed form for the star regular expression involves similar tricks for the sequence regular expression. The *Suffix* function is now replaced by something slightly more complex, because the growth pattern of star regular expressions' derivatives is a bit different:

$$\begin{array}{ll}
r^* & \longrightarrow_{\backslash c} \\
(r\backslash c) \cdot r^* & \longrightarrow_{\backslash c'} \\
r\backslash cc' \cdot r^* + r\backslash c' \cdot r^* & \longrightarrow_{\backslash c''} \\
(r_1\backslash cc'c'' \cdot r^* + r\backslash c'') + (r\backslash c'c'' \cdot r^* + r\backslash c'' \cdot r^*) & \longrightarrow_{\backslash c'''} \\
\ldots &
\end{array}$$

When we have a string $s = c :: c' :: c'' \ldots$ such that $r\backslash c$, $r\backslash cc'$, $r\backslash c'$, $r\backslash cc'c''$, $r\backslash c'c''$, $r\backslash c''$ etc. are all nullable, the number of terms in $r^*\backslash s$ will grow exponentially rather than linearly in the sequence case. An examples of this:

$$
\begin{array}{ll}
(a + aa + aaa)^* \backslash aaa & = \\
((\mathbf{1}) \cdot (a + aa + aaa)^* + (\mathbf{1} + a + aa) \cdot (a + aa + aaa)^*) + & \\
((\mathbf{1} + a) \cdot (a + aa + aaa)^* + (\mathbf{1} + a + aa) \cdot (a + aa + aaa)^*) &
\end{array}
$$

The point is that the function *rsimp* ignores the difference between different nesting patterns of alternatives, and the exponentially growing star derivative like

$$(r_1 \backslash cc'c'' \cdot r^* + r \backslash c'') + (r \backslash c'c'' \cdot r^* + r \backslash c'' \cdot r^*)$$

can be treated as

$$\sum [r_1 \backslash cc'c'' \cdot r^*, r \backslash c'', r \backslash c'c'' \cdot r^*, r \backslash c'' \cdot r^*].$$

which can be de-duplicated by *rdistinct* and therefore bounded finitely. The closed form of star regular expressions is the theorem for doing this. In other words, we define a suitable *rs* where

$$r^* \backslash_{rsimps} s = rsimp\ \sum rs.$$

holds. More specifically, the list *rs* shall be in the form of $map\ (\lambda s'.r \backslash s' \cdot r^*)\ Ss$. *Ss* is a list of strings, and for example in the sequence closed form it is specified as $Suffix\ s\ r_1$. To get *Ss* for the star regular expression, we need to introduce *starUpdate* and *starUpdates*:

$$
\begin{array}{lcl}
starUpdate\ c\ r\ [] & \stackrel{\text{def}}{=} & [] \\
starUpdate\ c\ r\ (s :: Ss) & \stackrel{\text{def}}{=} & \\
 & & \quad if\ (rnullable\ (r \backslash_{rs} s)) \\
 & & \quad then\ (s@[c]) :: [c] :: (starUpdate\ c\ r\ Ss) \\
 & & \quad else\ (s@[c]) :: (starUpdate\ c\ r\ Ss)
\end{array}
$$

$$
\begin{array}{lcl}
starUpdates\ []\ r\ Ss & = & Ss \\
starUpdates\ (c :: cs)\ r\ Ss & = & starUpdates\ cs\ r\ (starUpdate\ c\ r\ Ss)
\end{array}
$$

With *starUpdates* defined, the closed form for star regular expressions can be given as:

Theorem 7. $r^* \backslash_{rsimps} (c :: s) = rsimp\ (\sum (map\ (\lambda s.(r \backslash_{rsimps} s) \cdot r^*)\ (starUpdates\ s\ r\ [[c]])))$

We omit the (quite tedious) proof because it contains a similar set of rewrite steps as the proofs of the sequence closed form (6). For readers interested in the details, see the Isabelle formalisation [46] for details.

6.4 Bounding Closed Forms

In this section, we introduce how we proved the bound on derivatives $r \backslash_{rsimps} s$ by bounding the closed forms derivatives. We first show that the total size of $rdistinct\ rs\ \varnothing$ is bounded by a constant *C* which depends on the maximum size of any element of *rs* only, not on the list length of *rs*. Then we prove that functions such as *rflts* will not cause the size of r-regular expressions to grow. This leads to the lemma that *rsimp* does not increase the size of a regular expression.

6.4.1 Finiteness of Distinct Regular Expressions

We define the set of regular expressions whose size is no more than a certain size N as *sizeNregex N*:

$$sizeNregex\ N \stackrel{\text{def}}{=} \{r \mid [\![r]\!]_r \leq N\}$$

Then the following lemma holds:

Lemma 15. *If $\forall r \in rs$, $[\![r]\!]_r \leq N$, then there exists a natural number d_N such that $[\![rdistinct\ rs\ \varnothing]\!]_r \leq d_N$ holds.*

Proof. Let $d_N = c_N * N$, where the constant $c_N = card\ (sizeNregex\ N)$. For all r in *set* $(rdistinct\ rs\ \varnothing)$, it is always the case that $[\![r]\!]_r \leq N$. In addition, we know that *length rdistinct rs* $\varnothing \leq c_N$ as every element in *rs* is unique and with a size smaller than N. □

This fact will be handy in estimating the closed form sizes.

6.4.2 *rsimp* Does Not Increase the Size

Although intuitive, proving that $[\![rsimp\ r]\!]_r$ is less than or equal to $[\![r]\!]_r$ is harder than we expected.

Lemma 16.

- $[\![rsimp_{ALTS}\ rs]\!]_r \leq [\![\sum\ rs]\!]_r$
- $[\![rsimp_{SEQ}\ r_1\ r_2]\!]_r \leq [\![r_1 \cdot r_2]\!]_r$
- $[\![rflts\ rs]\!]_r \leq [\![rs]\!]_r$
- $[\![rdistinct\ rs\ ss]\!]_r \leq [\![rs]\!]_r$
- *If all elements a in the set as satisfy the property that $[\![rsimp\ a]\!]_r \leq [\![a]\!]_r$, then we have* $[\![\ rsimp_{ALTS}\ (rdistinct\ (rflts\ (map\ rsimpas))\{\})]\!] \leq$

  $[\![\ \sum\ (rdistinct\ (rflts\ (map\ rsimp\ x))\ \{\})]\!]_r$
- $[\![rsimp\ r]\!]_r \leq [\![r]\!]_r$

The sub-lemmas gradually build up until the last one. All proofs can be completed with suitable induction and automation and are therefore omitted.

6.4.3 Estimating the Closed Forms' sizes

We recap the closed forms we obtained earlier:

- $(\sum rs)\backslash_{rsimps} s \stackrel{\text{rsimp}}{=} \sum\ (map\ (\_\backslash_{rsimps} s)\ rs)$
- $(r_1 \cdot r_2)\backslash_{rsimps} s \stackrel{\text{rsimp}}{=} \sum((r_1\backslash s) \cdot r_2) :: (map\ (r_2\backslash\_)(Suffix\ s\ r_1))$
- $r^*\backslash_{rsimps} c :: s = rsimp\ (\sum(map\ (\lambda s.(r\backslash_{rsimps} s) \cdot r^*)\ (starUpdates\ s\ r\ [[c]])))$

The closed forms on the left-hand-side are all of the same shape: $rsimp\ (\sum rs)$. This expression, in turn is equal to $rsimp_{ALTS}\ (rdistinct\ rs'\ \varnothing)$ for some rs' and therefore the general bound for *rdistinct*'s output (lemma 15) applies.

We elaborate the above reasoning by a series of lemmas below, where straightforward proofs are omitted.

Lemma 17.

- *If we have three accumulator sets: noalts_set, alts_set and corr_set, satisfying:*
  - $\forall r \in noalts\_set.\ \nexists xs.\ r = \sum xs$
  - $\forall r \in alts\_set.\ \exists xs.\ r = \sum xs$ *and set* $xs \subseteq corr\_set$

  then we have that

$$\begin{array}{ll} [\![(rdistinct\ (rflts\ rs)\ (noalts\_set \cup corr\_set))]\!]_r & \leq \\ [\![(rdistinct\ rs\ (noalts\_set \cup alts\_set \cup \{\mathbf{0}_r\}))]\!]_r & \end{array}$$

  holds.

- *rdistinct and rflts working together is at least as good as rdistinct alone, which can be written as*

$$[\![rdistinct\ \ (rflts\ rs)\ \ \varnothing]\!]_r \leq [\![rdistinct\ \ rs\ \ \varnothing]\!]_r.$$

- $[\![rsimp\ \sum rs]\!]_r \leq [\![rdistinct\ \ rs\ \ \varnothing]\!]_r + 1$

Proof. The last part is directly from the second part. The second part is a direct corollary of the first by letting *noalts_set* and *alts_set* be equal to $\varnothing$, which gives us

$$[\![rdistinct\ \ (rflts\ rs)\ \ \varnothing]\!]_r \leq [\![rdistinct\ \ rs\ \ \{\mathbf{0}_r\}]\!]_r.$$

RHS is less than or equal to $[\![rdistinct\ \ rs\ \ \varnothing]\!]_r$.

We focus on the first sub-lemma. Here is the intuition behind the slightly weird preconditions: We split the accumulator into two parts: the part which contains alternative regular expressions (*alts_set*), and the part without any of them(*noalts_set*). The set *corr_set* is the corresponding set of *alts_set* with all elements under the alternative constructor spilled out.

Now we do an induction on the list *rs*. For the base case, the two terms are equal and therefore the inequality is vacuously true. For the inductive case, the inductive hypothesis is that

$$\begin{array}{ll} [\![(rdistinct\ (rflts\ rs)\ (noalts\_set \cup corr\_set))]\!]_r & \leq \\ [\![(rdistinct\ rs\ (noalts\_set \cup alts\_set \cup \{\mathbf{0}_r\}))]\!]_r & \end{array}$$

already holds. Now for $rs' = r :: rs$, the LHS is equal to

$$[\![(rdistinct\ (rflts\ (r :: rs))\ (noalts\_set \cup corr\_set))]\!]_r.$$

We do a case analysis on r. (i) If r is equal to $\mathbf{0}_r$ then the above is equal to

$$[\![(rdistinct\ (rflts\ (rs))\ (noalts\_set \cup corr\_set))]\!]_r.$$

which by the inductive hypothesis is less than or equal to

$$\begin{array}{ll} [\![(rdistinct\ rs\ (noalts\_set \cup alts\_set \cup \{\mathbf{0}_r\}))]\!]_r & = \\ [\![(rdistinct\ (\mathbf{0}_r :: rs)\ (noalts\_set \cup alts\_set \cup \{\mathbf{0}_r\}))]\!]_r & \end{array}$$

(ii) If r is not an alternative, LHS is equal to

$$[\![(rdistinct\ (r :: (rflts\ (rs)))\ (noalts\_set \cup corr\_set))]\!]_r.$$

in the case that $r \notin noalts\_set$, this is equal to

$$[\![r :: (rdistinct\ (rflts\ (rs))\ ((noalts\_set \cup \{r\}) \cup corr\_set))]\!]_r.$$

Now setting $noalts\_set' = (noalts\_set \cup \{r\})$, we are able to use the inductive hypothesis:

$$\begin{array}{ll}
[\![r :: (rdistinct\ (rflts\ (rs))\ ((noalts_set \cup \{r\}) \cup corr_set))]\!]_r & = \\
[\![r :: (rdistinct\ (rflts\ (rs))\ (noalts_set' \cup corr_set))]\!]_r & \leq \\
[\![r :: (rdistinct\ rs\ (noalts_set' \cup alts_set \cup \{\mathbf{0}_r\}))]\!]_r & = \\
[\![(rdistinct\ (r :: rs)\ (noalts_set \cup alts_set \cup \{\mathbf{0}_r\}))]\!]_r. & = \\
RHS. &
\end{array}$$

The last term is equal to the RHS of the inequality desired. Similar reasoning steps apply to the situation where $r \in noalts\_set$.
(iii) If r is an alternative, then we have that $r = \sum rs''$ for some rs''.

$$\begin{array}{ll}
LHS & = \\
[\![(rdistinct\ (rs''@rflts\ (rs))\ (noalts_set \cup corr_set))]\!]_r & \leq \\
[\![(rdistinct\ rs''@rs\ (noalts_set \cup corr_set))]\!]_r & = \\
[\![(rdistinct\ (\sum rs'')\ ::\ rs\ (noalts_set \cup corr_set))]\!]_r & = \\
[\![(rdistinct\ (r :: rs)\ (noalts_set \cup alts_set \cup \{\mathbf{0}_r\}))]\!]_r. & = \\
RHS. &
\end{array}$$

□

We are now ready to show the bound on the closed forms of $(r_1 \cdot r_2)\backslash s$ and $r^*\backslash s$ and in general $r\backslash s$.

Theorem 8. *For any regex r, one can pick a natural number N_r, s.t. $\forall s.\ [\![r\backslash_{rsimp} s]\!]_r \leq N_r$.*

Proof. We prove this by induction on r. The base cases for $\mathbf{0}_r$, $\mathbf{1}_r$ and $\mathbf{c}_r$ are straightforward. In the sequence $r_1 \cdot r_2$ case, the inductive hypotheses state $\exists N_1.\forall s.\ [\![r\backslash_{rsimp} s]\!] \leq N_1$ and $\exists N_2.\forall s.\ [\![r_2\backslash_{rsimp} s]\!] \leq N_2$.

When the string s is not empty, we can reason as follows

$$\begin{array}{lll}
 & [\![r_1 \cdot r_2\backslash_{rsimp} s]\!]_r & \\
= & [\![rsimp\ (\sum(r_1\backslash_{rsimp} s \cdot r_2\ ::\ map\ (r_2\backslash_{rsimp_})\ (Suffix\ s\ r)))]\!]_r & (1) \\
\leq & [\![rdistinct\ \ (r_1\backslash_{rsimp} s \cdot r_2\ ::\ map\ (r_2\backslash_{rsimp_})\ (Suffix\ s\ r))\ \ \varnothing]\!]_r + 1 & (2) \\
\leq & 2 + N_1 + [\![r_2]\!]_r + (N_2 * (card\ (sizeNregex\ N_2))) & (3)
\end{array}$$

(1) is by theorem 6. (2) is by 17. (3) is by 15.

Combining the cases when $s = []$ and $s \neq []$, we get (4):

$$[\![(r_1 \cdot r_2)\backslash_r s]\!]_r \quad \leq \quad max\ (2 + N_1 + [\![r_2]\!]_r + N_2 * (card\ (sizeNregex\ N_2)))\ [\![r_1 \cdot r_2]\!]_r \quad (4)$$

We reason similarly for $STAR$. The inductive hypothesis is $\exists N.\forall s.\ [\![r\backslash_{rsimp} s]\!] \leq N$. Let $n_r = [\![r^*]\!]_r$. When $s = c :: cs$ is not empty,

$$\begin{array}{lll}
 & [\![r^*\backslash_{rsimp} c :: cs]\!]_r & \\
= & [\![rsimp\ (\sum(map\ (\lambda s.(r\backslash_{rsimp} s) \cdot r^*)\ (starUpdates\ cs\ r\ [[c]])))]\!]_r & (5) \\
\leq & [\![rdistinct\ \ (map\ (\lambda s.(r\backslash_{rsimp} s) \cdot r^*)\ (starUpdates\ cs\ r\ [[c]]))\ \ \varnothing]\!]_r + 1 & (6) \\
\leq & 1 + (card(sizeNregex\ (N + n_r))) * (1 + (N + n_r)) & (7)
\end{array}$$

(5) is by theorem 7. (6) is by 17. (7) is by corollary 15. Combining with the case when $s = []$, one obtains

$$[\![r^*\backslash_r s]\!]_r \quad \leq \quad max\ n_r\ 1 + (card(sizeNregex\ (N + n_r))) * (1 + (N + n_r)) \quad (8)$$

The alternative case is slightly less involved. The inductive hypothesis is equivalent to $\exists N.\forall r \in (map\ (\_\backslash_r s)\ rs).[\![r]\!]_r \leq N$. In the case when $s = c :: cs$, we have

$$\begin{array}{lll} & [\![\sum rs \backslash_{rsimps} c :: cs]\!]_r & \\ = & [\![rsimp\ (\sum(map\ (\_\backslash_{rsimps} s)\ rs))]\!]_r & (9) \\ \leq & [\![(\sum(map\ (\_\backslash_{rsimps} s)\ rs))]\!]_r & (10) \\ \leq & 1 + N * (length\ rs) & (11) \end{array}$$

(9) is by theorem 5, (10) by lemma 16 and (11) by inductive hypothesis.

Combining with the case when $s = []$, we obtain

$$[\![\sum rs \backslash_r s]\!]_r \quad \leq \quad max\ [\![\sum rs]\!]_r\ 1 + N * (length\ rs) \quad (12)$$

We have all the inductive cases proven. □

We hope to improve this result in the future by making N_r a computable function on r. This would make the size being parametric on r clearer and provide a more useful estimate of the space complexity of *blexer_simp*. We also would like to improve the bound as the number N_r is quite large. We plan to make it polynomial on $[\![r]\!]_r$. Here is the main result on annotated regular expressions:

Corollary 4. *For any annotated regular expression a, $\exists N_r. \forall s.\ [\![a \backslash_{bsimps} s]\!] \leq N_r$*

Proof. By lemma 6, the size of annotated derivatives $[\![a \backslash_{bsimps} s]\!]$ can be expressed as the size of r-regular expressions' derivatives $[\![a_{\downarrow_r} \backslash_{rsimps} s]\!]_r$. Let $r = a_{\downarrow_r}$. Then by theorem 8, there exists N_r such that $[\![a\_ \downarrow_r \backslash_{rsimps} s]\!]_r \leq N_r$. □

6.5 Bounded Repetitions

We have promised in chapter 1 that our lexing algorithm can potentially be extended to handle bounded repetitions in natural and elegant ways. Now we fulfill our promise by adding support for the "exactly-n-times" bounded regular expression $r^{\{n\}}$. We add clauses in our derivatives-based lexing algorithms (with simplifications) introduced in chapter 5.

6.5.1 Augmented Definitions

There are a number of definitions that need to be augmented. The most notable one would be the POSIX rules for $r^{\{n\}}$. Again this has been formalised in [15], but we put it here nevertheless so the readers do not have to dig into the proof script:

$$\frac{\forall v \in vs_1. \vdash v : r \wedge |v| \neq [] \qquad \forall v \in vs_2. \vdash v : r \wedge |v| = [] \qquad length\ (vs_1 @ vs_2) = n}{Stars\ (vs_1 @ vs_2) : r^{\{n\}}}$$

As Ausaf had pointed out [13], sometimes empty iterations have to be taken to get a match with exactly n repetitions, and hence the vs_2 part.

Another important definition would be the size:

$$[\![r^{\{n\}}]\!]_r \quad \stackrel{\text{def}}{=} \quad [\![r]\!]_r + n$$

One could also choose $\log n$ or 1 for the constant factor of the right-hand-side. However for simplicity of the bound proof we choose to add the counter directly to the size.

For brevity, we sometimes use NTIMES to refer to bounded-repetition regular expressions. The derivative, *mkeps* and *inj* function clause for NTIMES are defined as (see [15]):

$$
\begin{array}{lcl}
r^{\{n\}}\backslash_r c & \stackrel{\text{def}}{=} & \begin{array}{ll} r\backslash_r c \cdot r^{\{n-1\}} & \text{if } n \geq 1 \\ \mathbf{0}_r & \text{otherwise} \end{array} \\
\mathit{mkeps}\ r^{\{n\}} & \stackrel{\text{def}}{=} & \mathit{Stars}\ (\mathit{replicate}\ n\ (\mathit{mkeps}\ r)) \\
\mathit{inj}\ r^{\{n\}}\ c\ (\mathit{Seq}\ v\ (\mathit{Stars}\ vs)) & \stackrel{\text{def}}{=} & \mathit{Stars}\ ((\mathit{inj}\ r\ c\ v) :: vs)
\end{array}
$$

6.5.2 Proofs for the Augmented Lexing Algorithm

We need to maintain two proofs with the additional $r^{\{n\}}$ construct: the correctness proof in chapter 5, and the finiteness proof in chapter 6.

Correctness Proof Augmentation

The correctness of *lexer* and *blexer* with bounded repetitions have been proven by Ausaf and Urban[14]. As they have commented, once the definitions are in place, the proofs given for the basic regular expressions will extend to bounded regular expressions, and there are no "surprises". We confirm this point because the correctness theorem would also extend without surprise to *blexer_simp*. The rewrite rules such as $\rightsquigarrow$ and $\overset{s}{\rightsquigarrow}$ do not involve STAR, and therefore do not involve STAR-like expressions like NTIMES either. The proof arguments remain largely unchanged except for minor modifications to the strucutral induction cases. Readers can look at the details in the Isabelle proof entry.

Finiteness Proof Augmentation

The NTIMES regular expressions are very similar to STAR, and therefore the treatment is similar, with minor changes to handle some slight complications when the counter reaches 0. The exponential growth is similar, assuming the counter does not reach 0 for the first few derivatives:

$$
\begin{array}{ll}
r^{\{n\}} & \longrightarrow_{\backslash c} \\
(r\backslash c) \cdot r^{\{n-1\}} & \longrightarrow_{\backslash c'} \\
r\backslash cc' \cdot r^{\{n-1\}} + r\backslash c' \cdot r^{\{n-2\}} & \longrightarrow_{\backslash c''} \\
(r_1\backslash cc'c'' \cdot r^{\{n-1\}} + r\backslash c'' \cdot r^{\{n-2\}}) + (r\backslash c'c'' \cdot r^{\{n-2\}} + r\backslash c'' \cdot r^{\{n-3\}}) & \longrightarrow_{\backslash c'''} \\
\ldots &
\end{array}
$$

Again, we assume that $r\backslash c$, $r\backslash cc'$ and so on are all nullable. We can flatten the nested alternative

$$(r_1\backslash cc'c'' \cdot r^{\{n-1\}} + r\backslash c'' \cdot r^{\{n-2\}}) + (r\backslash c'c'' \cdot r^{\{n-2\}} + r\backslash c'' \cdot r^{\{n-3\}}) + \ldots$$

into the list of terms for $r^{\{n\}}\backslash_{rs} s$

$$[r_1\backslash cc'c'' \cdot r^{\{n-1\}},\ r\backslash c'' \cdot r^{\{n-2\}},\ r\backslash c'c'' \cdot r^{\{n-2\}},\ r\backslash c'' \cdot r^{\{n-3\}},\ \ldots]$$

An example where $r = (a^*)^{\{3\}}$:

$$\begin{array}{lcl}
(a^*)^{\{3\}}\backslash aa & \stackrel{\text{rsimp}}{=} & a^* \cdot r^{\{2\}} + a^* \cdot r^{\{1\}} \\
(a^*)^{\{3\}}\backslash aaa & \stackrel{\text{rsimp}}{=} & a^* \cdot r^{\{2\}} + a^* \cdot r^{\{1\}} + a^* \cdot r^{\{1\}} + a^* \cdot r^{\{0\}} \\
(a^*)^{\{3\}}\backslash aaaa & \stackrel{\text{rsimp}}{=} & a^* \cdot r^{\{2\}} + a^* \cdot r^{\{1\}} + a^* \cdot r^{\{1\}} + a^* \cdot r^{\{0\}} + a^* \cdot r^{\{1\}} + a^* \cdot r^{\{0\}} + \mathbf{0}_r
\end{array}$$

The growth rate was 2 in each step except for the last one, where the expression with counter 0 is not expanded by turned into $\mathbf{0}_r$. Just like what we did for STAR, we prove the closed form

$$r^{\{n\}}\backslash_{rsimps} s = rsimp\ \sum rs.$$

for some regular expression list rs. In the setting of NTIMES, the *starUpdate* and *starUpdates* functions are replaced by *nupdate* and *nupdates*:

$$\begin{array}{lcl}
nupdate & :: & "char \Rightarrow rrexp \Rightarrow ((string, nat)\ option)\ list \Rightarrow rrexp" \\
nupdate\ c\ r\ [] & \stackrel{\text{def}}{=} & [] \\
nupdate\ c\ r\ (Some\ (s,\ n+1)\ ::\ Ss) & \stackrel{\text{def}}{=} & if\ (rnullable\ \ (r\backslash_{rs} s)) \\
 & & \quad then\ \ Some\ (s@[c], n+1) :: Some\ ([c], n) :: (nupdate\ c\ r\ Ss) \\
 & & \quad else\ \ Some\ (s@[c], n+1) :: (nupdate\ c\ r\ Ss) \\
nupdate\ c\ r\ (None :: Ss) & \stackrel{\text{def}}{=} & (None :: nupdate\ c\ r\ Ss)
\end{array}$$

$$\begin{array}{lcl}
nupdates & :: & "string \Rightarrow rrexp \Rightarrow ((string, nat)\ option)\ list \Rightarrow rrexp" \\
nupdates\ []\ r\ Ss & \stackrel{\text{def}}{=} & Ss \\
nupdates\ (c :: cs)\ r\ Ss & \stackrel{\text{def}}{=} & nupdates\ cs\ r\ (nupdate\ c\ r\ Ss)
\end{array}$$

which takes into account when a subterm $(r\backslash_s s \cdot r^{\{n\}})$'s counter is 1, and therefore expands to

$$r\backslash_s(s@[c]) \cdot r^{\{n\}}\ +\ \mathbf{0}$$

after taking a derivative. The elements of the argument Ss now have type

$$(string, nat)\ option$$

and therefore the function for converting such an element into a regular expression term is called *opterm*:

$$\begin{array}{lcl}
opterm\ r\ SN & \stackrel{\text{def}}{=} & case\ SN\ of \\
 & & \quad Some\ (s, n) \Rightarrow (r\backslash_{rs} s) \cdot r^{\{n\}} \\
 & & \quad None \Rightarrow \mathbf{0}.
\end{array}$$

The list rs is instantiated as

$$map\ (opterm\ r)\ (nupdates\ s\ r\ [Some\ ([c], n)]).$$

We also define the simplified version of *opterm*, which is *optermsimp*:

$$\begin{array}{lcl}
optermsimp\ r\ SN & \stackrel{\text{def}}{=} & case\ SN\ of \\
 & & \quad Some\ (s, n) \Rightarrow (r\backslash_{rsimps} s) \cdot r^{\{n\}} \\
 & & \quad None \Rightarrow \mathbf{0}
\end{array}$$

Now we are ready to present the closed form for NTIMES:

Theorem 9. *(The closed form for bounded-repetition regular expression)*

$$r^{\{n+1\}}\backslash_{rsimps}(c :: s) = rsimp\ (\sum(map\ (optermsimp\ r)\ (nupdates\ s\ r\ [Some\ ([c], n)]))).$$

The proof is by a structural induction on the string s, and omitted here (for details see [46]). The key observation for bounding this closed form is that the counter on $r^{\{n\}}$ will only decrement during derivatives:

Lemma 18.

- *For an element o in set $(nupdates\ s\ r\ [Some\ ([c], n)])$, either $o = None$, or $o = Some\ (s', m)$ for some string s' and number $m \leq n$.*
- *For any element r' in set $(map\ (optermsimp\ r)\ (nupdates\ s\ r\ [Some\ ([c], n)]))$, we have that r' is either $\mathbf{0}$ or $r\backslash_r s' \cdot r^{\{m\}}$ for some string s' and number $m \leq n$.*

The proof is routine and therefore omitted. But this lemma is crucial in establishing the bound, as without them we do not know why $rdistinct\ \ rs\ \varnothing$ would be finite for a list rs like

$$(map\ (optermsimp\ r)\ (nupdates\ s\ r\ [Some\ ([c], n)])).$$

Theorem 10. *Assuming that for any string s, $[\![r\backslash_{rsimps} s]\!]_r \leq N$ holds, then we have that $[\![r^{\{n+1\}}\backslash_{rsimps} s]\!]_r \leq max\ (c_N + 1) * (N + [\![r^{\{n\}}]\!] + 1)$, where $c_N = card\ (sizeNregex\ (N + [\![r^{\{n\}}]\!]_r + 1))$.*

Proof. We have that for all regular expressions r' in

$$set\ (map\ (optermsimp\ r)\ (nupdates\ s\ r\ [Some\ ([c], n)])),$$

r''s size is less than or equal to $N + [\![r^{\{n\}}]\!]_r + 1$ because r' can only be either a $\mathbf{0}$ or $r\backslash_{rsimps} s' \cdot r^{\{m\}}$ for some string s' and number $m \leq n$ (lemma 18). In addition, we know that the list $map\ (optermsimp\ r)\ (nupdates\ s\ r\ [Some\ ([c], n)])$'s size is at most $c_N = card\ (sizeNregex\ ((N + [\![r^{\{n\}}]\!]) + 1))$. This gives us $[\![r\backslash_{rsimps}\ s]\!]_r \leq N * c_N$. □

We aim to formalise the correctness and size bound for constructs like $r^{\{\ldots n\}}$, $r^{\{n\ldots\}}$ and so on, which is still work in progress. They should more or less follow the same recipe described in this section. Once we know how to deal with them recursively using suitable auxiliary definitions, we can establish the proofs with a more or less automated procedure.

6.6 Comments and Future Improvements

6.6.1 Some Experimental Results

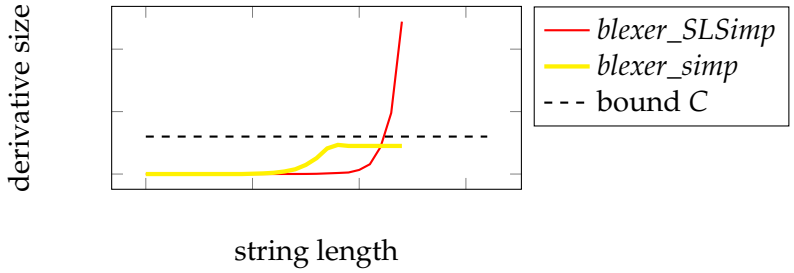
What guarantee does this bound give us? It states that given a regular expression r, its derivative with any string is bounded by a constant number that only depends on r, not s. In other words, there does not exist a regular expression r and an infinite list of strings $S = [s_1, s_2, \ldots]$ such that for any natural number n, a string s_i can be picked from S to make $[\![r\backslash_{rsimps} s_i]\!]$ greater than n. Sulzmann and Lu's original optimised lexer does not have such a property. For example, one can pick $r = (a + aa)^*$ and $S = [a, aa, \ldots]$ and get a series of ever-growing derivatives. Abusing mathematical notation, we might say that for Sulzmann and Lu's lexer, some r can cause the derivative size to go to infinity:

$$\lim_{i \to \infty} [\![blexer\_SLSimp\ r\ s_i]\!] = \infty$$

But for our lexer, all regular expression r has an associated constant C such that

$$\lim_{i \to \infty} [\![blexer\_simp\ r\ s_i]\!] \leq C.$$

We show the picture at the beginning of this chapter again:



In our proof for the inductive case $r_1 \cdot r_2$, the dominant term in the bound is $l_{N_2} * N_2$, where N_2 is the bound we have for $[\![r_2 \backslash_{bsimps} s]\!]$. Given that l_{N_2} is roughly the size 4^{N_2}, the size bound $[\![r_1 \cdot r_2 \backslash_{bsimps} s]\!]$ inflates the size bound of $[\![r_2 \backslash_{bsimps} s]\!]$ with the function $f(x) = x * 2^x$. This means the bound we have will probably surge up at least tower-exponentially with a linear increase of the depth. One might be pretty skeptical about what this non-elementary bound can bring us.

We admit that in order to get a more straightforward proof we sacrificed the precision of our estimate of the closed forms, getting us a very loose bound. However, having a constant bound means we can already treat each derivative operation as constant time (except for bitcodes). Though in theory the constant factor can still be quite large, in practice this rarely happens. Most of the regex's sizes seem to stay within a polynomial bound *w.r.t* the original size. For Sulzmann and Lu's simplified lexer this is not the case. We will discuss detailed improvements to this bound in the next chapter.

6.6.2 Possible Further Improvements

There are two problems with this finiteness result, though:

- First, it is not yet a direct formalisation of our lexer's complexity, as a complexity proof would require looking into the time it takes to execute **all** the operations involved in the lexer (simp, collect, decode), not just the derivative.
- Second, the bound is not yet tight, and we seek to improve C so that it is a polynomial of $[\![a]\!]$.

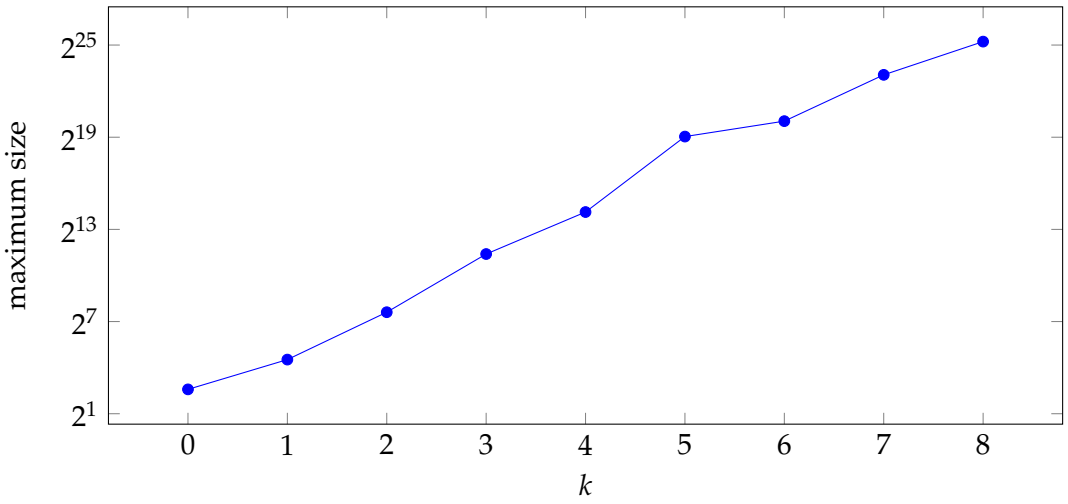
Still, we believe this bound is useful, because

- The size proof can serve as a starting point for a complexity formalisation. Derivatives are the most important phases of our lexer algorithm. Size properties about derivatives cover the majority of the algorithm and is therefore a good indication of the complexity of the entire program.
- The bound is already a strong indication that catastrophic backtracking is much less likely to occur in our *blexer_simp* algorithm. We refine *blexer_simp* with *blexerStrong* in the next chapter so that we conjecture the bound becomes polynomial.

One might wonder if there are "evil" regular expressions for our *blexer_simp*. The most "evil" regular expression we found is the following:

$$(a^* + (aa)^* + (aaa)^* + \ldots + (\underbrace{a \ldots a}_{\text{k a's}})^*)^*$$

Even though it seems quite unnatural and unlikely to appear in practice. For convenience we use $(\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{i a's}})^*)^*$ to express $(a^* + (aa)^* + (aaa)^* + \ldots + (\underbrace{a \ldots a}_{\text{n a's}})^*)$ in the below discussion. The regular expression $(\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{i a's}})^*)^*$ grows quite a bit at the beginning when taking derivatives against the string of as. The growth will eventually stop, with a maximum size that is exponential on the number k:



The exponential size is triggered by that the regex $\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{i a's}})^*$ inside the $(\ldots)^*$ having exponentially many different derivatives, despite those differences being minor. $(\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{i a's}})^*)^* \backslash \underbrace{a \ldots a}_{\text{m a's}}$ will therefore contain the following terms (after flattening out all nested alternatives):

$$(\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{((i - (m' \% i))\%i) a's}}) \cdot (\underbrace{a \ldots a}_{\text{i a's}})^*) \cdot (\sum_{i=1}^{n}(\underbrace{a \ldots a}_{\text{i a's}})^*)$$
$$(1 \leq m' \leq m)$$

There are at least exponentially many such terms.[3] With each new input character taking the derivative against the intermediate result, more and more such distinct terms will accumulate. The function *distinctBy* will not be able to de-duplicate any two of these terms

$$(\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{((i - (m' \% i))\%i) a's}}) \cdot (\underbrace{a \ldots a}_{\text{i a's}})^*) \cdot (\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{i a's}})^*)^*$$
$$(\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{((i - (m'' \% i))\%i) a's}}) \cdot (\underbrace{a \ldots a}_{\text{i a's}})^*) \cdot (\sum_{i=1}^{k}(\underbrace{a \ldots a}_{\text{i a's}})^*)^*$$

where $m' \neq m''$ as they are slightly different. For example, even if having a lot of overlapping subterms, the expression

$$(a^* + (a) \cdot (aa)^* + (aa) \cdot (aaa)^*) \cdot (\sum_{i=1}^{3}(\underbrace{a \ldots a}_{\text{i a's}})^*)^*$$

[3] To be exact, these terms are distinct for $m' \leq L.C.M.(1, \ldots, n)$, a detailed mathematical analysis is omitted, but the point is that the number is exponential.

and

$$(a^* + (a) \cdot (aa)^* + (aaa)^*) \cdot (\sum_{i=1}^{3} (\underbrace{a \ldots a}_{\text{i a's}})^*)^*$$

are deemed different, and no simplification applies between those two elements on a list *rs* when simplified by *distinctBy*. This is because *blexer_simp* does not have a distrbutivity rewrite rule. If this rule was added, the output value would no longer be POSIX.

This means that with our current simplification methods, we will not be able to control the derivative so that $[\![r \backslash_{bsimps} s]\!]_r$ stays polynomial with respect to $[\![r]\!]_r$.

One viable simplification rule for the above example might be $(a + b) \cdot r + (a + c) \cdot r = (a + b) \cdot r + c \cdot r$. This does not break the POSIXness of *blexer_simp*, though it is more cumbersome to define. This example also suggests a slightly different notion of size, which we call the alphabetic width:

$$\begin{array}{lcl}
awidth\ \mathbf{0} & \stackrel{\text{def}}{=} & 0 \\
awidth\ \mathbf{1} & \stackrel{\text{def}}{=} & 0 \\
awidth\ c & \stackrel{\text{def}}{=} & 1 \\
awidth\ r_1 + r_2 & \stackrel{\text{def}}{=} & awidth\ r_1 + awidth\ r_2 \\
awidth\ r_1 \cdot r_2 & \stackrel{\text{def}}{=} & awidth\ r_1 + awidth\ r_2 \\
awidth\ r^* & \stackrel{\text{def}}{=} & awidth\ r
\end{array}$$

Antimirov[10] has proven that $PDER_{UNIV}(r) \leq awidth(r)$, where $PDER_{UNIV}(r)$ is a set of all possible subterms created by doing derivatives of r against all possible strings. If we can make sure that at any moment in our lexing algorithm our intermediate result hold at most one copy of each of the subterms then we can get the same bound as Antimirov's. This leads to the algorithm in the next chapter.

Chapter 7

A Better Size Bound for Derivatives

This chapter is a "work-in-progress" chapter which records extensions to our *blexer_simp*. We make a conjecture that the finite size bound from the previous chapter can be improved to a cubic bound. We implemented our conjecture in Scala. We have not formalised this part in Isabelle/HOL. Nevertheless, we outline the ideas we intend to use for the proof.

7.1 A Stronger Version of Simplification

Let us first present further improvements for our lexer algorithm *blexer_simp*. We devise a stronger simplification algorithm, called *bsimpStrong*, which can prune away similar components in two regular expressions at the same alternative level, even if these regular expressions are not exactly the same. We call the lexer that uses this stronger simplification function *blexerStrong*. We conjecture that both

$$blexerStrong\ r\ s = blexer\ r\ s$$

and

$$[\![a \backslash_{bsimpStrongs} s]\!] = O([\![a]\!]^3)$$

hold. We give an informal justification why the correctness and cubic size bound proofs can be achieved by exploring the connection between the internal data structure of our *blexerStrong* and Animirov's partial derivatives.

In our bitcoded lexing algorithm, (sub)terms represent (sub)matches. For example, the regular expression

$$aa \cdot a^* + a \cdot a^* + aa \cdot a^*$$

contains three terms, expressing three possibilities for how it can match some more input of the form $a \ldots a$. The first and the third terms are identical, which means we can eliminate the latter as it will not contribute to a POSIX value. In *bsimp*, the *distinctBy* function takes care of such instances. The criteria *distinctBy* uses for removing a duplicate a_2 in the list

$$rs_a @ [a_1] @ rs_b @ [a_2] @ rs_c$$

is that the two erased regular expressions are equal

$$(a_1)_{\downarrow_r} = (a_2)_{\downarrow_r}.$$

This is characterised as the *LD* rewrite rule in figure 5.4. The problem, however, is that identical components in two slightly different regular expressions cannot be removed by the *LD* rule. Consider the stronger simplification

$$(a + b + d) \cdot r_1 + (a + c + e) \cdot r_1 \overset{?}{\rightsquigarrow} (a + b + d) \cdot r_1 + (c + e) \cdot r_1 \tag{7.1}$$

where the $(\underline{a} + c + e) \cdot r_1$ is deleted in the right alternative $a + c + e$. This is permissible because we have $(a + \ldots) \cdot r_1$ in the left alternative. The difficulty is that such "buried" alternatives-sequences are not easily recognised. But simplification like this actually cannot be omitted, if we want to have a better bound. For example, the size of derivatives can still blow up even with our *bsimp* function: consider again the example $((a^* + (aa)^* + \ldots + (\underbrace{\underline{a} \ldots \underline{a}}_{na's})^*)^*)^*$, and set n to a relatively small number like $n = 5$, then we get the following exponential growth:

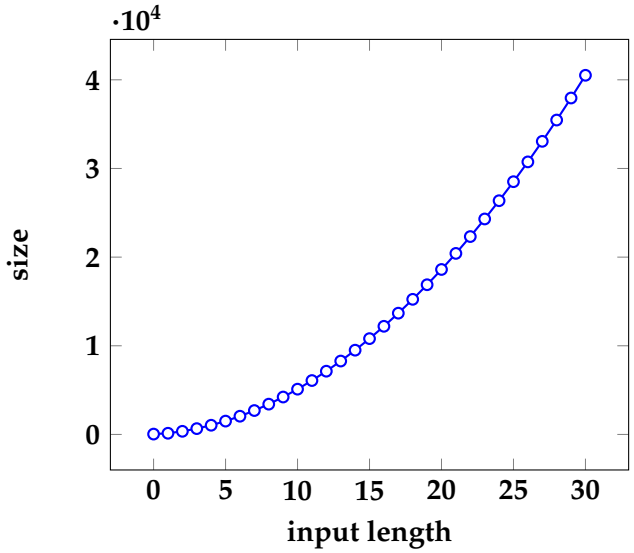


FIGURE 7.1: Size of derivatives of *blexer_simp* from chapter 5 for matching $((a^* + (aa)^* + \ldots + (aaaaa)^*)^*)^*$ with strings of the form $\underbrace{aa..a}_{n}$.

One possible approach would be to apply the rewriting rule

$$(a + b + d) \cdot r_1 \longrightarrow a \cdot r_1 + b \cdot r_1 + d \cdot r_1$$

which pushes the sequence into the alternatives in our *simp* function. This would then make the simplification shown in (7.1) possible. Translating this rule into our *bsimp* function would simply involve adding a new clause to the $bsimp_{ASEQ}$ function:

$$\begin{array}{lcl} bsimp_{ASEQ}\ bs\ a\ b & \stackrel{\text{def}}{=} & (a,\ b) match \\ & & \ldots \\ & & \quad case\ (_{bs1}\sum as, a_2') \Rightarrow_{bs1} \sum(map\ (_{[]}ASEQ\ \_\ a_2')\ as) \\ & & \quad case\ (a_1', a_2') \Rightarrow_{bs}\ a_1' \cdot a_2' \end{array}$$

Unfortunately, if we introduce this clause in our setting we would lose the POSIX property of our calculated values. For example given the regular expression

$$(a + ab)(bc + c)$$

and the string ab, then our algorithm generates the following correct POSIX value

$$Seq\ (Right\ ab)\ (Right\ c).$$

Essentially it matches the string with the longer Right-alternative in the first sequence (and then the 'rest' with the character regular expression c from the second sequence). If we add the simplification above, however, then we would obtain the following value

$$Left\ (Seq\ a\ (Left\ bc))$$

where the *Left*-alternatives get priority. This violates the POSIX rules. The reason for getting this undesired value is that the new rule splits this regular expression up into a topmost alternative

$$a \cdot (bc + c) + ab \cdot (bc + c),$$

which is a regular expression with a quite different meaning: the original regular expression is a sequence, but the simplified regular expression is an alternative. With an alternative the maximal munch rule no longer works.

A method to reconcile this problem is to do the transformation in (7.1) "non-invasively", meaning that we traverse the list of regular expressions inside alternatives

$$\sum(rs_a @ [a] @ rs_c)$$

using a function similar to *distinctBy*, but this time we allow the following more general rewrite rule:

$$\frac{}{rs_a @ [a] @ rs_c \overset{s}{\rightsquigarrow} rs_a @ [prune\ a\ rs_a] @ rs_c}\ \text{CUBICRULE} \tag{7.2}$$

where *prune a acc* traverses a without altering the structure of a, but removing components in a that have appeared in the accumulator *acc*. For example

$$prune \quad (r_a + r_f + r_g + r_h)r_d \quad [(r_a + r_b + r_c)r_d, (r_e + r_f)r_d]$$

should be equal to

$$(r_g + r_h)r_d$$

because $r_g r_d$ and $r_h r_d$ are the only terms that do not appeared in the accumulator list

$$[(r_a + r_b + r_c)r_d, (r_e + r_f)r_d].$$

We implemented the function *prune* in Scala (see figure 7.2) The function *prune* is a stronger version of *distinctBy*. It does not just walk through a list looking for exact duplicates, but prunes sub-expressions recursively. It manages proper contexts by the helper functions *removeSeqTail*, *isOne* and *atMostEmpty*.
Suppose we feed

$$r = (\underline{\mathbf{1}} + (\underline{f} + b) \cdot g) \cdot (a \cdot (d \cdot e))$$

and

$$acc = \{a \cdot (d \cdot e), f \cdot (g \cdot (a \cdot (d \cdot e)))\}$$

```
1   def prune(r: ARexp, acc: Set[Rexp]) : ARexp = r match{
2    case AALTS(bs, rs) => rs.map(r => prune(r, acc)).filter(_ !=
        AZERO) match
3    {
4     //all components have been removed, meaning this is
         effectively a duplicate
5     //flats will take care of removing this AZERO
6     case Nil => AZERO
7     case r::Nil => fuse(bs, r)
8     case rs1 => AALTS(bs, rs1)
9    }
10   case ASEQ(bs, r1, r2) =>
11    //remove the r2 in (ra + rb)r2 to identify the duplicate
         contents of r1
12    prune(r1, acc.map(r => removeSeqTail(r, erase(r2)))) match {
13     //after pruning, returns 0
14     case AZERO => AZERO
15     //after pruning, got r1'.r2, where r1' is equal to 1
16     case r1p if(isOne(erase(r1p))) => fuse(bs ++ mkepsBC(r1p),
          r2)
17     //assemble the pruned head r1p with r2
18     case r1p => ASEQ(bs, r1p, r2)
19    }
20    //this does the duplicate component removal task
21   case r => if(acc(erase(r))) AZERO else r
22  }
```

FIGURE 7.2: The function *prune* is called recursively in the alternative case (line 2) and in the sequence case (line 12). In the alternative case we keep all the accumulators the same, but in the sequence case we are making necessary changes to the accumulators to allow correct de-duplication.

```
def atMostEmpty(r: Rexp) : Boolean = r match {
  case ZERO => true
  case ONE => true
  case STAR(r) => atMostEmpty(r)
  case SEQ(r1, r2) => atMostEmpty(r1) && atMostEmpty(r2)
  case ALTS(r1, r2) => atMostEmpty(r1) && atMostEmpty(r2)
  case CHAR(_) => false
}

def isOne(r: Rexp) : Boolean = r match {
  case ONE => true
  case SEQ(r1, r2) => isOne(r1) && isOne(r2)
  case ALTS(r1, r2) => (isOne(r1) || isOne(r2)) &&
     (atMostEmpty(r1) && atMostEmpty(r2))
  case STAR(r0) => atMostEmpty(r0)
  case CHAR(c) => false
  case ZERO => false
}

def removeSeqTail(r: Rexp, tail: Rexp) : Rexp =
  if (r == tail)
    ONE
  else {
    r match {
      case SEQ(r1, r2) =>
        if(r2 == tail) r1 else ZERO
      case r => ZERO
    }
  }
```

FIGURE 7.3: The helper functions of *prune*: *atMostEmpty*, *isOne* and *removeSeqTail*. *atMostEmpty* is a function that takes a regular expression and returns true only in case that it contains nothing more than the empty string in its language. *isOne* tests whether a regular expression is equivalent to **1**. *removeSeqTail* is a function that takes away the tail of a sequence regular expression.

as the input into *prune*. The end result will be

$$b \cdot (g \cdot (a \cdot (d \cdot e)))$$

where the underlined components in r are eliminated. Looking more closely, at the topmost call

$$prune \quad (\mathbf{1} + (f + b) \cdot g) \cdot (a \cdot (d \cdot e)) \quad \{a \cdot (d \cdot e), f \cdot (g \cdot (a \cdot (d \cdot e)))\}$$

The sequence clause will be called, where a sub-call

$$prune \;\; (\mathbf{1} + (f + b) \cdot g) \;\; \{\mathbf{1}, f \cdot g\}$$

is made. The terms in the new accumulator $\{\mathbf{1},\ f \cdot g\}$ come from the two calls to *removeSeqTail*:

$$removeSeqTail \qquad a \cdot (d \cdot e) \qquad a \cdot (d \cdot e)$$

and

$$removeSeqTail \qquad f \cdot (g \cdot (a \cdot (d \cdot e))) \qquad a \cdot (d \cdot e).$$

The idea behind *removeSeqTail* is that when pruning recursively, we need to "zoom in" to sub-expressions, and this "zoom in" needs to be performed on the accumulators as well, otherwise the deletion will not work. The sub-call $prune \;\; (\mathbf{1} + (f + b) \cdot g) \;\; \{\mathbf{1}, f \cdot g\}$ is simpler, which will trigger the alternative clause, causing a pruning on each element in $(\mathbf{1} + (f + b) \cdot g)$, leaving us with $b \cdot g$ only.

Our new lexer with stronger simplification uses *prune* by making it the core component of the deduplicating function called *distinctWith*. *DistinctWith* ensures that all verbose parts of a regular expression are pruned away.

```
def turnIntoTerms(r: Rexp): List[Rexp] = r match {
 case SEQ(r1, r2) =>
  turnIntoTerms(r1).flatMap(r11 => furtherSEQ(r11, r2))
   case ALTS(r1, r2) => turnIntoTerms(r1) :::
      turnIntoTerms(r2)
   case ZERO => Nil
   case _ => r :: Nil
}

def distinctWith(rs: List[ARexp],
 pruneFunction: (ARexp, Set[Rexp]) => ARexp,
 acc: Set[Rexp] = Set()) : List[ARexp] =
  rs match{
   case Nil => Nil
   case r :: rs =>
    if(acc(erase(r)))
     distinctWith(rs, pruneFunction, acc)
    else {
     val pruned_r = pruneFunction(r, acc)
     pruned_r ::
     distinctWith(rs,
      pruneFunction,
      turnIntoTerms(erase(pruned_r)) ++: acc
      )
    }
  }
```

FIGURE 7.4: A Stronger Version of *distinctBy*. This function allows "partial de-duplication" of a regular expression. When part of a regular expression has appeared before in the accumulator, we can remove that verbose component.

Once a regular expression has been pruned, all its components will be added to the accumulator to remove any future regular expressions' duplicate components.

The function *bsimpStrong* is very much the same as *bsimp*, just with *distinctBy* replaced by *distinctWith*.

```
def bsimpStrong(r: ARexp): ARexp =
{
  r match {
    case ASEQ(bs1, r1, r2) => (bsimpStrong(r1), bsimpStrong(r2))
       match {
    case (AZERO, _) => AZERO
    case (_, AZERO) => AZERO
    case (AONE(bs2), r2s) => fuse(bs1 ++ bs2, r2s)
    case (r1s, AONE(bs2)) => fuse(bs1, r1s) //assert bs2 == Nil
    case (r1s, r2s) => ASEQ(bs1, r1s, r2s)
    }
    case AALTS(bs1, rs) => {
     distinctWith(flats(rs.map(bsimpStrong(_))), prune) match {
       case Nil => AZERO
       case s :: Nil => fuse(bs1, s)
       case rs => AALTS(bs1, rs)
     }
    }
    case ASTAR(bs, r0) if(atMostEmpty(erase(r0))) => AONE(bs)
    case r => r
  }
}
def bdersStrong(s: List[Char], r: ARexp) : ARexp = s match {
    case Nil => r
    case c::s => bdersStrong(s, bsimpStrong(bder(c, r)))
  }
```

FIGURE 7.5: The function *bsimpStrong*: a stronger version of *bsimp*

The benefits of using *prune* refining the finiteness bound to a cubic bound has not been formalised yet. Therefore we choose to use Scala code rather than an Isabelle-style formal definition like we did for *bsimp*, as the definitions might change to suit our proof needs.

We conjecture that the above Scala function *bdersStrong*, written $\_\backslash_{bsimpStrongs}\_$ as an infix notation, satisfies the following property:

Conjecture 1. $[\![a\backslash_{bsimpStrongs}s]\!] = O([\![a]\!]^3)$

The stronger version of *blexer_simp*'s code in Scala looks like:

```
def strongBlexer(r: Rexp, s: String) : Option[Val] = {
  Try(Some(decode(r, strong_blex_simp(internalise(r),
      s.toList)))).getOrElse(None)
}
def strong_blex_simp(r: ARexp, s: List[Char]) : Bits = s
    match {
  case Nil => {
    if (bnullable(r)) {
      mkepsBC(r)
    }
    else
      throw new Exception("Not matched")
  }
  case c::cs => {
    strong_blex_simp(strongBsimp(bder(c, r)), cs)
  }
}
```

We call this lexer *blexerStrong*. This version is able to reduce the size of the derivatives which otherwise triggered exponential behaviour in *blexer_simp*. Consider again the runtime for matching $((a^* + (aa)^* + \ldots + (aaaaa)^*)^*)^*$ with strings of the form $\underbrace{aa..a}_{n}$. They produce the folloiwng graphs (*blexerStrong* on the left-hand-side and *blexer_simp* on the right-hand-side).

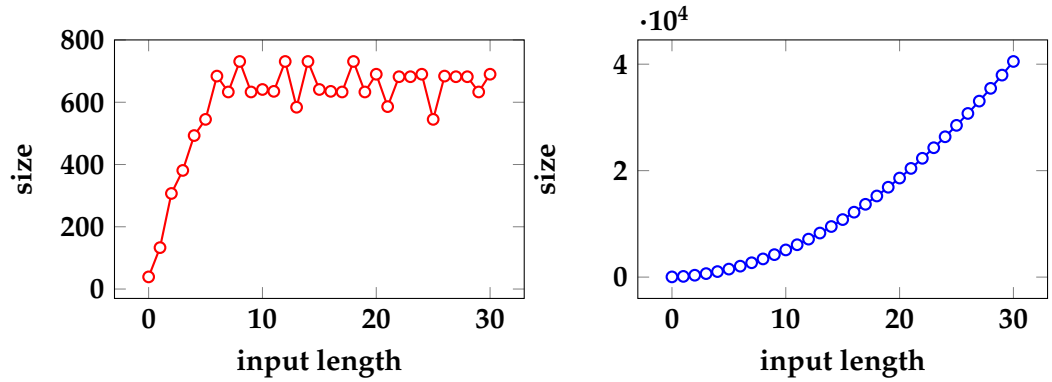


FIGURE 7.6

We hope the correctness is preserved. The proof idea is to preserve the key lemma in chapter 5 such as in equation (7.2).

Conjecture 2. *blexerStrong r s* = *blexer r s*

The idea is to maintain key lemmas in chapter 5 like $r \overset{*}{\rightsquigarrow} bsimp\ r$ with the new rewriting rule shown in figure (7.2) .

In the next sub-section, we will describe why we believe a cubic size bound can be achieved with the stronger simplification. For this we give a short introduction to the partial derivatives, which were invented by Antimirov [10], and then link them with the result of the function *bdersStrong*.

7.1.1 Antimirov's partial derivatives

Partial derivatives were first introduced by Antimirov [10]. They are very similar to Brzozowski derivatives, but split children of alternative regular expressions into multiple independent terms. This means the output of partial derivatives is a set of regular expressions, defined as follows

$$\begin{array}{lcll} \partial_x\ (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & (\partial_x\ r_1) \cdot r_2 \cup \partial_x\ r_2 & \textit{if nullable } r_1 \\ & & (\partial_x\ r_1) \cdot r_2 & \textit{otherwise} \\ \partial_x\ r^* & \stackrel{\text{def}}{=} & (\partial_x\ r) \cdot r^* & \\ \partial_x\ c & \stackrel{\text{def}}{=} & \textit{if } x = c \textit{ then } \{\mathbf{1}\} \textit{ else } \varnothing & \\ \partial_x(r_1 + r_2) & = & \partial_x(r_1) \cup \partial_x(r_2) & \\ \partial_x(\mathbf{1}) & = & \varnothing & \\ \partial_x(\mathbf{0}) & \stackrel{\text{def}}{=} & \varnothing & \end{array}$$

The $\cdot$ in the example $(\partial_x\ r_1) \cdot r_2$ is a shorthand notation for the cartesian product $(\partial_x\ r_1) \times \{r_2\}$. Rather than joining the calculated derivatives $\partial_x r_1$ and $\partial_x r_2$ together using the $\sum$ constructor, Antimirov put them into a set. This means many subterms will be de-duplicated because they are sets. For example, to compute what the derivative of the regular expression $x^*(xx + y)^*$ w.r.t. x is, one can compute a partial derivative and get two singleton sets $\{x^* \cdot (xx + y)^*\}$ and $\{x \cdot (xx + y)^*\}$ from $\partial_x(x^*) \cdot (xx + y)^*$ and $\partial_x((xx + y)^*)$.

The partial derivative w.r.t. a string is defined recursively:

$$\partial_{c::cs} r \stackrel{\text{def}}{=} \bigcup_{r' \in (\partial_c r)} \partial_{cs} r'$$

Suppose an alphabet Σ, we use Σ^* for the set of all possible strings from the alphabet. The set of all possible partial derivatives is then defined as the union of derivatives w.r.t all the strings:

$$PDER_{\Sigma^*}\ r \stackrel{\text{def}}{=} \bigcup_{w \in \Sigma^*} \partial_w\ r$$

Consider now again our pathological case where we apply the more aggressive simplification

$$((a^* + (aa)^* + \ldots + (\underbrace{a \ldots a}_{n a's})^*)^*)^*$$

let use abbreviate theis regular expression with r, then we have that

$$PDER_{\Sigma^*}\ r = \bigcup_{i=1}^{n} \bigcup_{j=0}^{i-1} \{(\underbrace{a \ldots a}_{\text{j a's}} \cdot (\underbrace{a \ldots a}_{\text{i a's}})^*) \cdot r\},$$

The union on the right-hand-side has $n * (n + 1)/2$ terms. This leads us to believe that the maximum number of terms needed in our derivative is also only $n * (n + 1)/2$. Therefore we conjecture that *bsimpStrong* is also able to achieve this upper limit in general.

Conjecture 3. *Using a suitable transformation f, we have that*

$$\forall s.\ f\ (r \setminus_{bsimpStrongs} s) \subseteq PDER_{\Sigma^*}\ r$$

holds.

The reason is that our (7.2) will keep only one copy of each term, where the function *prune* takes care of maintaining a set like structure similar to partial derivatives.

Antimirov had proven that the sum of all the partial derivative terms' sizes is bounded by the cubic of the size of that regular expression:

Property 9. $[\![PDER_{\Sigma^*}\ r]\!] \leq O([\![r]\!]^3)$

This property was formalised by Wu et al. [86], and the details can be found in the Archive of Formal Froofs[1]. Once conjecture 3 is proven, then property 9 would provide us with a cubic bound for our *blexerStrong* algorithm:

Conjecture 4. $[\![r \setminus_{bsimpStrongs} s]\!] \leq [\![r]\!]^3$

We leave this as future work.

[1] https://www.isa-afp.org/entries/Myhill-Nerode.html

Chapter 8

Conclusion and Future Work

In this thesis, in order to solve the ReDoS attacks once and for all, we have set out to formalise the correctness proof of Sulzmann and Lu's lexing algorithm with aggressive simplifications [77]. We formalised our proofs in the Isabelle/HOL theorem prover. We have fixed some inefficiencies and a bug in their original simplification algorithm, and established the correctness by proving that our algorithm outputs the same result as the original bit-coded lexer without simplifications (whose correctness was established in previous work by Ausaf et al. in [13] and [14]). The proof technique used in [13] does not work in our case because the simplification function messes with the structure of simplified regular expressions. Despite having to try out several workarounds and being stuck for months looking for proofs, we were delighted to have discovered the simple yet effective proof method by modelling individual simplification steps as small-step rewriting rules and proving equivalence between terms linked by these rewrite rules.

In addition, we have proved a formal size bound on the regular expressions. The technique was by establishing a "closed form" informally described by Murugesan and Shanmuga Sundaram [61] for compound derivatives and using those closed forms to control the size. The Isabelle/HOL code for our formalisation can be found at

`https://github.com/hellotommmy/posix`

Thanks to our theorem-prover-friendly approach, we believe that this finiteness bound can be improved to a bound linear to input and cubic to the regular expression size using a technique by Antimirov[10]. Once formalised, this would be a guarantee for the absence of all super-linear behavious. We are yet to work out the details.

Our formalisation is approximately 7500 lines of code. Slightly more than half of this concerns the finiteness bound obtained in Chapter 5. This slight "bloat" is because we had to repeat many definitions for the rrexp datatype. However, we think we would not have been able to formalise the quite intricate reasoning involving rrexps with annotated regular expressions because we would have to carry around the bit-sequences (that are of course necessary in the algorithm) but which do not contribute to the size bound of the calculated derivatives.

To our best knowledge, no lexing libraries using Brzozowski derivatives have similar complexity-related bounds, and claims about running time are usually speculative and backed up only by empirical evidence on some test cases. If a matching or lexing algorithm does not come with complexity related guarantees (for examaple the internal data structure size does not grow indefinitely), then one cannot claim with confidence of having solved the problem of catastrophic backtracking.

We believe our proof method is not specific to this algorithm, and intend to extend this approach to prove the correctness of the faster version of the lexer proposed in chapter [**Cubic**]. The closed forms can then be re-used as well. Together with the

idea from Antimirov [10] we plan to reduce the size bound to be just polynomial with respect to the size of the regular expressions.

We have learned quite a few lessons in this process. As simple as the end result may seem, coming up with the initial proof idea with rewriting relations was the hardest for us, as it required a deep understanding of what information is preserved during the simplification process. There the ideas given by Sulzmann and Lu and Ausaf et al. were of no use. Of course this has already been shown many times, the usefulness of formal approaches cannot be overstated: they not only allow us to find bugs in Sulzmann and Lu's simplification functions, but also helped us set up realistic expectations about performance of algorithms. We believed in the beginning that the *blexer_simp* lexer defined in chapter 5 was already able to achieve the linear bound claimed by Sulzmann and Lu. We then attempted to prove that the size $[\![blexer\_simp\ r\ s]\!]$ is $O([\![r]\!]^c \cdot |s|)$, where c is a small constant, making $[\![r]\!]^c$ a small constant factor. We were then quite surprised that this did not go through despite a lot of effort. This led us to discover the example where $[\![r]\!]^c$ can in fact be exponentially large in chapter 6. We therefore learned the necessity to introduce the stronger simplifications in chapter 7. Without formal proofs we would not have found out this so soon, if at all.

8.1 Future Work

This thesis is just a starting point for the line of research it opens up. We would have wanted *blexer_simp*'s correctness proof (and finite bound) to have come out sooner. Ideally this thesis's formal proofs should also be extended to handle more aggressive simplifications such as those in chapter 7. The actual difficulty of the proofs in chapter 5 and 6 turned out to be much more involved and therefore time-consuming than anticipated. We also wanted to greatly increase the usefulness of our lexer by adding support for back-references, accompanied by a formal semantics and formal correctness proofs, and we have already had some initial results. Unfortunately these results could not be put into this thesis due to the time limit we have.

The most obvious next-step is to implement the cubic bound and correctness of *blexerStrong* in chapter 7. A cubic bound ($O([\![r]\!]^c \cdot |s|)$) with respect to regular expression size will get us one step closer to fulfilling the linear complexity claim made by Sulzmann and Lu.

With a linear size bound theoretically, the next challenge would be to generate code that is competitive with respect to matchers based on DFAs or NFAs. For that a lot of optimisations are needed. We aim to integrate the zipper data structure into our lexer. The idea is very simple: using a zipper with multiple focuses just like Darragh [28] did in their parsing algorithm, we could represent

$$x \cdot r + y \cdot r + \ldots$$

as

$$(x + y + \ldots) \cdot r.$$

This would greatly reduce the amount of space needed when we have many terms like $x \cdot r$. Some initial results have been obtained, but significant changes are needed to make sure that the lexer output conforms to the POSIX standard. We aim to make use of Okui and Suzuki's labelling system [65] to ensure regular expressions represented as zippers always maintain the POSIX orderings.

To further optimise the algorithm, we plan to add a deduplicate function that tells whether two regular expressions denote the same language using an efficient and verified equivalence checker like [50]. In this way, the internal data structures can be pruned more aggressively, leading to better simplifications and ultimately faster algorithms.

Traditional automata approaches can be sped up by compiling multiple rules into the same automaton. This has been done in [53] and [18], for example. Currently our lexer only deals with a single regular expression each time. Extending this to multiple regular expressions might open up more possibilities of simplifications.

As already mentioned in chapter 1, reducing the number of memory accesses can also accelerate the matching speed. It would be interesting to study the memory bandwidth of our derivative-based matching algorithm and improve accordingly.

Memoization has been used frequently in lexing and parsing to achieve better complexity results, for example in [70], [33], [6], [28] and [30]. We plan to explore the performance enhancements by memoization in our algorithm in a correctness-preserving way. The monadic data refinement technique that Lammich and Tuerk used in [80] to optimise Hopcroft's automaton minimisation algorithm seems also quite relevant for such an enterprise. We aim to learn from their refinement framework which generates high performance code with proofs that can be broken down into small steps.

Extending the Sulzmann and Lu's algorithm to parse more pcre regex constructs like lookahead, capture groups and back-references with proofs on basic properties like correctness seems useful, despite it cannot be made efficient. Creating a correct and easy-to-prove version of *blexer_simp* seems an appealing next-step for both practice and theory.

Bibliography

[1] https://www.cloudflare.com/en-gb/. [Online; last accessed 22-Aug-2023.

[2] https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/. [Online; last accessed 22-Aug-2023.

[3] https://zherczeg.github.io/sljit/regex_perf.html.

[4] http://davidvgalbraith.com/how-i-fixed-atom/. [Online; last accessed 22-Aug-2023.

[5] In:

[6] M.D. Adams, C. Hollenbeck, and M. MightMatthew. "On the complexity and performance of parsing with derivatives". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2016, pp. 224–236.

[7] A.V. Aho. "CHAPTER 5 - Algorithms for Finding Patterns in Strings". In: *Algorithms and Complexity*. Ed. by JAN VAN LEEUWEN. Handbook of Theoretical Computer Science. Amsterdam: Elsevier, 1990, pp. 255–300. ISBN: 978-0-444-88071-0. DOI: https://doi.org/10.1016/B978-0-444-88071-0.50010-2. URL: https://www.sciencedirect.com/science/article/pii/B9780444880710500102.

[8] V Alfred. "Algorithms for finding patterns in strings". In: *Algorithms and Complexity* 1 (2014), p. 255.

[9] J. B. Almeida et al. "Partial Derivative Automata Formalized in Coq". In: *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA)*. Vol. 6482. LNCS. 2010, pp. 59–68.

[10] V. Antimirov. "Partial Derivatives of Regular Expressions and Finite Automata Constructions". In: *Theoretical Computer Science* 155 (1995), pp. 291–319.

[11] Robert Atkey. "Amortised resource analysis with separation logic". In: *European Symposium on Programming*. Springer. 2010, pp. 85–103.

[12] *Atom Editor*. https://github.com/atom/atom. [Online; last accessed 22-Aug-2023]. 2023.

[13] F. Ausaf. "Verified Lexing and Parsing". PhD thesis. King's College London, 2018.

[14] 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)*. Vol. 9807. LNCS. 2016, pp. 69–86.

[15] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. "POSIX Lexing with Derivatives of Regular Expressions". In: *Archive of Formal Proofs* (2016). https://isa-afp.org/entries/Posix-Lexing.html, Formal proof development. ISSN: 2150-914x.

[16] Jeremy Avigad and Kevin Donnelly. "Formalizing O notation in Isabelle/HOL". In: *International Joint Conference on Automated Reasoning*. Springer. 2004, pp. 357–371.

[17] L. Carettoni B. Caller. *regexploit*. 2021. URL: https://github.com/doyensec/regexploit.

[18] M. Becchi and Patrick. "Extending finite automata to efficiently match Perl-compatible regular expressions". In: Jan. 2008, p. 25. DOI: 10.1145/1544012.1544037.

[19] M. Becchi and P. Crowley. "An Improved Algorithm to Accelerate Regular Expression Evaluation". In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*. ANCS '07. Orlando, Florida, USA: Association for Computing Machinery, 2007, 145–154. ISBN: 9781595939456. DOI: 10.1145/1323548.1323573. URL: https://doi.org/10.1145/1323548.1323573.

[20] M. Berglund, W. Bester, and B. Van Der Merwe. "Formalising Boost POSIX Regular Expression Matching". In: *Theoretical Aspects of Computing – ICTAC 2018*. Ed. by Bernd Fischer and Tarmo Uustalu. Cham: Springer International Publishing, 2018, pp. 99–115. ISBN: 978-3-030-02508-3.

[21] Jan Bessai, Jakob Rehof, and Boris Düdder. "Fast Verified BCD Subtyping". In: June 2019, pp. 356–371. ISBN: 978-3-030-22347-2. DOI: 10.1007/978-3-030-22348-9_21.

[22] H .Björklund, W. Martens, and T. Timm. "Efficient Incremental Evaluation of Succinct Regular Expressions". In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM '15. Melbourne, Australia: Association for Computing Machinery, 2015, pp. 1541–1550. ISBN: 9781450337946. DOI: 10.1145/2806416.2806434. URL: https://doi.org/10.1145/2806416.2806434.

[23] J. A. Brzozowski. "Derivatives of Regular Expressions". In: *Journal of the ACM* 11.4 (1964), pp. 481–494.

[24] C. Câmpeanu, K. Salomaa, and S. Yu. "A formal study of practical regular expressions". In: *International Journal of Foundations of Computer Science* 14.06 (2003), pp. 1007–1018.

[25] C. Câmpeanu and N. Santean. "On the closure of pattern expressions languages under intersection with regular languages". In: *Acta Inf.* 46 (May 2009), pp. 193–207. DOI: 10.1007/s00236-009-0090-y.

[26] C. Câmpeanu and N. Santean. "On the intersection of regex languages with regular languages". In: *Theoretical Computer Science* 410.24 (2009). Formal Languages and Applications: A Collection of Papers in Honor of Sheng Yu, pp. 2336–2344. ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2009.02.022. URL: https://www.sciencedirect.com/science/article/pii/S0304397509001789.

[27] T. Coquand and V. Siles. "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. 2011, pp. 119–134.

[28] P. Darragh and M.D. Adams. "Parsing with Zippers (Functional Pearl)". In: *Proc. ACM Program. Lang.* 4.ICFP (2020). DOI: 10.1145/3408990. URL: https://doi.org/10.1145/3408990.

[29] J. C. Davis et al. “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)*. 2018, pp. 246–256.

[30] R. Edelmann. “Efficient Parsing with Derivatives and Zippers”. In: (2021), p. 246. DOI: 10.5075/epfl-thesis-7357. URL: http://infoscience.epfl.ch/record/287059.

[31] R. Edelmann, H. Jad, and K. Viktor. “Zippy LL(1) Parsing with Derivatives”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 1036–1051. ISBN: 9781450376136. DOI: 10.1145/3385412.3385992. URL: https://doi.org/10.1145/3385412.3385992.

[32] D. Egolf, S. Lasser, and K. Fisher. “Verbatim: A Verified Lexer Generator”. In: *2021 IEEE Security and Privacy Workshops (SPW)*. 2021, pp. 92–100. DOI: 10.1109/SPW53761.2021.00022.

[33] D. Egolf, S. Lasser, and K. Fisher. “Verbatim++: Verified, Optimized, and Semantically Rich Lexing with Derivatives”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2022. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 27–39. ISBN: 9781450391825. DOI: 10.1145/3497775.3503694. URL: https://doi.org/10.1145/3497775.3503694.

[34] H. Fernau and M.L. Schmid. “Pattern matching with variables: A multivariate complexity analysis”. In: *Information and Computation* 242 (2015), pp. 287–305. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2015.03.006. URL: https://www.sciencedirect.com/science/article/pii/S0890540115000218.

[35] D. Firas. *regex101*. 2011. URL: https://regex101.com/.

[36] G. Fowler. “An interpretation of the POSIX regex standard”. In: *URL: https://web.archive. org/web/20050408073627/http://www. research. att. com/~ gsf/testregex/reinterpretation. html* (2003).

[37] D.D. Freydenberger. “Extended Regular Expressions: Succinctness and Decidability”. In: *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*. Ed. by Thomas Schwentick and Christoph Dürr. Vol. 9. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 507–518. ISBN: 978-3-939897-25-5. DOI: 10.4230/LIPIcs.STACS.2011.507. URL: http://drops.dagstuhl.de/opus/volltexte/2011/3039.

[38] D.D. Freydenberger and M.L. Schmid. “Deterministic regular expressions with back-references”. In: *Journal of Computer and System Sciences* 105 (2019), pp. 1–39. ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2019.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0022000018301818.

[39] A. Frisch and L. Cardelli. “Greedy Regular Expression Matching”. In: *Proc. of the 31st International Conference on Automata, Languages and Programming (ICALP)*. Vol. 3142. LNCS. 2004, pp. 618–629.

[40] *GNU grep*. URL: https://www.gnu.org/software/grep/manual/grep.html.

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

[42] Armaël Guéneau, Arthur Charguéraud, and François Pottier. "A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification". In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 533–560. ISBN: 978-3-319-89884-1.

[43] P. Hazel. *PCRE*. 2021. URL: https://www.pcre.org/original/doc/html/.

[44] G. Huet. "The Zipper". In: *J. Funct. Program.* 7.5 (1997), 549–554. ISSN: 0956-7968. DOI: 10.1017/S0956796897002864. URL: https://doi.org/10.1017/S0956796897002864.

[45] *Isabelle Formalisation of this thesis*. https://yurichev.com/news/20200621_regex_SAT/. [Online; last accessed 11-September-2023]. 2022.

[46] *Isabelle Formalisation of this thesis*. https://github.com/hellotommmy/posix/tree/main. [Online; last accessed 11-September-2023]. 2022.

[47] C. Doczkaland J.O. Kaiser and G. Smolka. "A Constructive Theory of Regular Languages in Coq". In: *Proceedings of the Third International Conference on Certified Programs and Proofs - Volume 8307*. Berlin, Heidelberg: Springer-Verlag, 2013, 82–97. ISBN: 9783319035444. DOI: 10.1007/978-3-319-03545-1_6. URL: https://doi.org/10.1007/978-3-319-03545-1_6.

[48] SC Kleene. "REPRESENTATION OF EVENTS IN NERVE NETS AND FINITE AUTOMATA[1]". In: *Automata Studies: Annals of Mathematics Studies. Number 34* 34 (1956), p. 3.

[49] S.C. Kleene et al. "Representation of events in nerve nets and finite automata". In: *Automata studies* 34 (1956), pp. 3–41.

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

[51] Alexander Krauss and Tobias Nipkow. "Regular Sets and Expressions". In: *Archive of Formal Proofs* (2010). https://isa-afp.org/entries/Regular-Sets.html, Formal proof development. ISSN: 2150-914x.

[52] C. Kuklewicz. *Regex Posix*. 2017. URL: https://wiki.haskell.org/Regex_Posix.

[53] S. Kumar et al. "Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection". In: *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '06. Pisa, Italy: Association for Computing Machinery, 2006, 339–350. ISBN: 1595933085. DOI: 10.1145/1159913.1159952. URL: https://doi.org/10.1145/1159913.1159952.

[54] Fritz Henglein Lasse Nielsen. "Bit-coded Regular Expression Parsing". In: *LATA* (2011).

[55] Michael E. Lesk and Eric E. Schmidt. "Lex—a lexical analyzer generator". In: 1990. URL: https://api.semanticscholar.org/CorpusID:7900881.

[56] F. Drewes M. Berglund and B. Van Der Merwe. "Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching". In: *Electronic Proceedings in Theoretical Computer Science* 151 (May 2014). DOI: 10.4204/EPTCS.151.7.

[57] M. Might, D. Darais, and D. Spiewak. "Parsing with Derivatives: A Functional Pearl". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. Tokyo, Japan: Association for Computing Machinery, 2011, pp. 189–195. ISBN: 9781450308656. DOI: 10.1145/2034773.2034801. URL: https://doi.org/10.1145/2034773.2034801.

[58] Yasuhiko Minamide, Yuto Sakuma, and Andrei Voronkov. "Translating regular expression matching into transducers". English. In: *Proceedings - 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2010 | Proc. - Int. Symp. Symb. Numer. Algorithms Sci. Comput., SYNASC*. 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2010 ; Conference date: 01-07-2011. 2011, pp. 107–115. ISBN: 9780769543246. DOI: 10.1109/SYNASC.2010.50.

[59] T. Miyazaki and Y. Minamide. "Derivatives of Regular Expressions with Lookahead". In: *Journal of Information Processing* 27 (2019), pp. 422–430. DOI: 10.2197/ipsjjip.27.422.

[60] Dan Moseley et al. "Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics". In: *Proc. ACM Program. Lang.* 7.PLDI (2023). DOI: 10.1145/3591262. URL: https://doi.org/10.1145/3591262.

[61] N. Murugesan and O. V. Shanmuga Sundaram. "Some Properties of Brzozowski Derivatives of Regular Expressions". In: *International Journal of Computer Trends and Technology* 13.1 (2014), pp. 29–33. URL: http://arxiv.org/abs/1407.5902.

[62] T. Nipkow. "Verified Lexical Analysis". In: *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. Vol. 1479. LNCS. 1998, pp. 1–15.

[63] T. Nipkow. "Verified Lexical Analysis". In: *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings*. Ed. by Jim Grundy and Malcolm C. Newey. Vol. 1479. Lecture Notes in Computer Science. Springer, 1998, pp. 1–15. DOI: 10.1007/BFb0055126. URL: https://doi.org/10.1007/BFb0055126.

[64] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media, 2002.

[65] S. Okui and T. Suzuki. "Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions". In: *Implementation and Application of Automata*. Ed. by Michael Domaratzki and Kai Salomaa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 231–240. ISBN: 978-3-642-18098-9.

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

[67] L. C. Paulson. "A Formalisation of Finite Automata Using Hereditarily Finite Sets". In: *Proc. of the 25th International Conference on Automated Deduction (CADE)*. Vol. 9195. LNAI. 2015, pp. 231–245.

[68] V. Paxson. "Bro: A System for Detecting Network Intruders in Real-Time". In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, p. 3.

[69] A. Rathnayake and H. Thielecke. “Static Analysis for Regular Expression Exponential Runtime via Substructural Logics”. In: *ArXiv* abs/1405.7058 (2014).

[70] T. Reps. ““Maximal-Munch” Tokenization in Linear Time”. In: *ACM Trans. Program. Lang. Syst.* 20.2 (1998), 259–273. ISSN: 0164-0925. DOI: 10.1145/276393.276394. URL: https://doi.org/10.1145/276393.276394.

[71] R. Ribeiro and A. D. Bois. “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, 2017. ISBN: 9781450353892. DOI: 10.1145/3125374.3125381. URL: https://doi.org/10.1145/3125374.3125381.

[72] M. Roesch. “Snort - Lightweight Intrusion Detection for Networks”. In: *Proceedings of the 13th USENIX Conference on System Administration*. LISA ’99. Seattle, Washington: USENIX Association, 1999, pp. 229–238.

[73] J. Sakarovitch. *Elements of Automata Theory*. Ed. by ReubenTranslator Thomas. Cambridge University Press, 2009. DOI: 10.1017/CBO9781139195218.

[74] M.L. Schmid. “Inside the Class of REGEX Languages”. In: *Proceedings of the 16th International Conference on Developments in Language Theory*. DLT’12. Taipei, Taiwan: Springer-Verlag, 2012, pp. 73–84. ISBN: 9783642316524. DOI: 10.1007/978-3-642-31653-1_8. URL: https://doi.org/10.1007/978-3-642-31653-1_8.

[75] R. Sidhu and V.K. Prasanna. “Fast Regular Expression Matching Using FPGAs”. In: *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’01)*. 2001, pp. 227–238.

[76] *Snort Community Rules*. https://www.snort.org/faq/what-are-community-rules. [Online; last accessed 19-November-2022]. 2022.

[77] M. Sulzmann and K. Lu. “POSIX Regular Expression Parsing with Derivatives”. In: *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*. Vol. 8475. LNCS. 2014, pp. 203–220.

[78] M. Sulzmann and P. van Steenhoven. “A Flexible and Efficient ML Lexer Tool Based on Extended Regular Expression Submatching”. In: *Proc. of the 23rd International Conference on Compiler Construction (CC)*. Vol. 8409. LNCS. 2014, pp. 174–191.

[79] M. Sulzmann and P. Thiemann. “Derivatives for Regular Shuffle Expressions”. In: *Proc. of the 9th International Conference on Language and Automata Theory and Applications (LATA)*. Vol. 8977. LNCS. 2015, pp. 275–286.

[80] P. Lammichand T. Tuerk. “Applying Data Refinement for Monadic Programs to Hopcroft’s Algorithm”. In: Aug. 2012, pp. 166–182. ISBN: 978-3-642-32346-1. DOI: 10.1007/978-3-642-32347-8_12.

[81] L. Turoňová et al. “Regex matching with counting-set automata”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–30. DOI: 10.1145/3428286. URL: https://doi.org/10.1145%2F3428286.

[82] L. Turoňová et al. “Regex Matching with Counting-Set Automata”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020). DOI: 10.1145/3428286. URL: https://doi.org/10.1145/3428286.

[83] Leslie G. Valiant. “General Context-Free Recognition in Less than Cubic Time”. In: *J. Comput. Syst. Sci.* 10 (1975), pp. 308–315.

[84] S. Vansummeren. "Type Inference for Unique Pattern Matching". In: *ACM Transactions on Programming Languages and Systems* 28.3 (2006), pp. 389–428.

[85] N. Weideman. "Static analysis of regular expressions". In: 2017.

[86] C. Wu, X. Zhang, and C. Urban. "A Formalisation of the Myhill-Nerode Theorem based on Regular Expressions". In: *Journal of Automatic Reasoning* 52.4 (2014), pp. 451–480.

[87] H. Chenand S. Yu. "Derivatives of Regular Expressions and an Application". In: vol. 7160. Jan. 2012, pp. 343–356. DOI: 10.1007/978-3-642-27654-5_27.