Verified Lexing and Parsing

Syed Fahad Ausaf Jafri

Department of Informatics

King's College London

A thesis submitted for the degree of

Doctor of Philosophy in Computer Science

September 2018

This thesis is dedicated to

my wonderful mother Rubina and my lovely fiancee Saher

for their unconditional love and support.

Acknowledgements

I would like to thank my supervisor, Christian Urban, for his unconditional support, for his time and useful conversations, for his guidance and advice, for going an extra mile and dealing with the concerned departments in troubled times and for backing me up during the course of my PhD.

I would also like to thank Antoine Delignat-Lavaud, Samin Ishtiaq and Nuno Lopes from Microsoft Research Cambridge for teaching me TLS protocol and LLVM, and for their constant mentoring and career counselling.

I am very grateful to my friends, Ehsan Ghoreishi and Umair Saleem, for their continuous support, for hearing me out all the time, for mentoring and for rescuing me over and over again.

My fiance, Saher Mirza, and my brother, Naveed Ausaf, deserve special mention for being my savior and support system throughout. I would also like to thank my family, especially my mother, Rubina Javed, for her unconditional love and support. Last but not the least, I gratefully acknowledge my friends, Owais Ahmed and Xin Jin, and my lab colleagues, Sumayyah, Jonathan Cardoso, Liana Mari-

nescu, Tomas Vitek, Tsvetan Jivkov, Pablo De Castro, Xin Lu, and especially Umesh Kumar, for their cooperation and assistance.

Declarations

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified below and in the text.

Some of the results in Chapters 2 and 3 have been published in earlier in a joint conference paper together with Dr Roy Dyckhoff from St Andrews University and Dr Christian Urban from King's College London. The results presented in Chapter 4 and 5 about TLS Message Parsers are a product of collaborations with Dr Antoine Delignat-Lavaud from the Programming Principles and Tools Group at Microsoft Research Cambridge, and Tej Chadej from The Electrical Engineering and Computer Science Department at MIT.

This dissertation contains fewer than 150,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Fahad Ausaf

September 2018

Abstract

The main part of this thesis is about regular expression matching. We shall focus on a POSIX lexer introduced by Sulzmann and Lu, which uses Brzozowski's Derivatives of Regular Expressions. These derivatives can be used for a simple regular expression matching algorithm. Sulzmann and Lu cleverly extended this algorithm in order to deal with POSIX matching, which is the underlying disambiguation strategy for regular expressions needed in lexers. Their algorithm generates POSIX values which encode information for *how* a regular expression matches a string–that is, which part of the string is matched by which part of the regular expression. We shall give our own inductive definition of what a POSIX value is and show that such a value is unique (for given regular expression and string being matched) and that Sulzmann and Lus algorithm always generates such a value (provided that the regular expression matches the string). We also show that our inductive definition of a POSIX value is equivalent to an alternative definition by Okui and Suzuki which identifies POSIX values as least elements according to an ordering of values. We also prove the correctness of Sulzmann and Lu's bitcoded version of the POSIX matching algorithm and extend the results to additional constructors for regular expressions.

In the second part, we focus on the specification of parsers for the Transport Layer Security (TLS) protocol. We have proved in $F^{\star}$

language the correctness and security of the parsers' pure specifications and derive efficient (zero-copy) and composable implementations from these specifications. The F* code is then extracted to C-code using the recent tactics engine of F*. For this, we also created a new library providing a unified model for bytes, replacing the previous unsound library. We then updated the TLS parsers to use this new byte model and enhance the functionality and verification automation.

Contents

I POSIX Regular Expression Matching 12

1 POSIX Regular Expression Matching 13

1.1 Introduction 14

1.2 POSIX Lexing 25

1.3 Preliminaries 30

1.4 POSIX Lexing Algorithm by Sulzmann and Lu 36

2 Specification of POSIX Values 46

2.1 Our POSIX Definition 46

2.2 Ordering of Values According to Okui and Suzuki 51

2.3 GREEDY Ordering by Frisch and Cardelli 60

2.4 POSIX Ordering by Sulzmann and Lu 62

3 Optimisations, Extensions and Future Work 67

3.1 Simplification of Regular Expressions 68

3.2 Bitcoded Values and Annotated Regular Expressions 75

3.3 Extensions 86

3.4 Summary and Future Work 92

II TLS Message Parsers 96

4 Project Everest 97

4.1 Introduction . . . 98
4.2 The HTTPS Ecosystem . . . 98
4.3 Project Everest . . . 101
4.3.1 The Everest Toolchain . . . 101
4.3.2 The Everest Runtime . . . 104

5 A Pure Model of Bytes 107

5.1 TLS Message Parsers . . . 107
5.2 Correctness Specifications for Parsers . . . 109
5.3 The New F* Bytes Library . . . 110
5.3.1 Parser . . . 111
5.4 Summary . . . 115

III Appendixes 117

A Bytes Library for TLS Message Parsers 118

Bibliography 131

List of Figures

1.1 The lexing algorithm by Sulzmann & Lu 42

2.1 Our inductive definition of POSIX values. 47

2.2 The reflexive version of the ordering by Frisch and Cardelli for GREEDY matching. 60

2.3 The reflexive version of the ordering by Sulzmann and Lu for POSIX matching. 63

3.1 Auxiliary functions for simplifying regular expressions and rectifying values. 70

4.1 A rough overview over the HTTPS ecosystem given by the Everest Project. 100

4.2 An overview over the Everest toolchain. 102

4.3 Overview over the reference implementation of TLS, called MiTLS. 103

4.4 Everest runtime: left is the functional runtime and right is low-level runtime. 104

5.1 A simple TLS datatype structure. 107

5.2 TLS parser variable length data structure. 108

5.3 The TLS low-level parsing framework. 109

Listings

5.1 Bytes append function and lemmas. 112

5.2 Bytes subtract, index, and length functions and lemmas. 112

5.3 Transform and concatenate a natural number to bytes. 113

5.4 Parsing variable length fields. 114

5.5 Finite Field Diffie-Hellman group definitions. 115

Part I

POSIX Regular Expression Matching

Chapter 1

POSIX Regular Expression Matching

This part is about regular expression matching using derivatives of regular expressions. These derivatives have been introduced by Brzozowski in 1964 in a paper where he showed that they can be used for a very simple regular expression matching algorithm [15]. The material presented in this part is mainly based on a paper by Sulzmann and Lu [53] published in 2014. Their paper introduces a clever extension of Brzozowski's algorithm which, in cases where a regular expression matches a string, calculates also a value for indicating *how* the regular expression matched the string. Such a value is important when one wants to know which substring is matched by which part of the regular expression, or when one wants to extract substrings from a larger string. It is also important for lexers that need to tokenise input strings. Our main contribution in this part are Isabelle proofs for establishing the correctness of Sulzmann and Lu's regular expression matching algorithm. The paper by Sulzmann and Lu already presents some "pencil-and-paper" proofs for the correctness, but these informal proofs contain some, which we believe, unfillable gaps and even errors—some of the

errors are already acknowledged by the authors in the online version of their paper.[1] To formally prove in Isabelle/HOL the correctness of the algorithm by Sulzmann and Lu, we introduce our own inductive definition of what a POSIX value is. We also provide formalised proofs for some of the unproven claims by Sulzmann and Lu about bitcoded regular expression matching [53]. Our work draws upon earlier work by Vansummeren [58], and Okui and Suzuki [41]. In fact we show that our own definition for POSIX values is equivalent to the one introduced by Okui and Suzuki. The Isabelle/HOL code of our formalisation is available from

`https://github.com/fahadausaf/POSIX-Parsing`

The results from Chapter 1, as well as from Sections 2.1 and 3.1 are also in the Archive of Formal Proofs of Isabelle.[2]

1.1 Introduction

Regular expressions are extremely useful for many text-processing tasks, such as finding substrings in large texts, lexing programs, syntax highlighting and so on. They also play a central role in security related programs, such as Snort and Bro [46, 49]. These programs employ sometimes thousands of regular expressions in order to find suspicious patterns in hostile network traffic. Since even small servers can nowadays handle large volumes of network traffic, fast regular expression matchers have become part of the critical computing infrastructure.

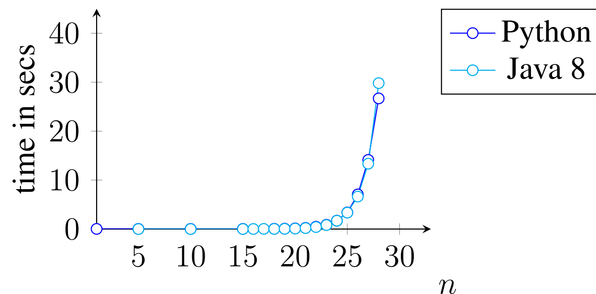
Given that regular expressions were introduced by Kleene in 1950 [30], one might think regular expressions have since been studied and implemented to

[1] `http://www.home.hs-karlsruhe.de/~suma0002/publications/regex-parsing-derivatives.pdf`, see for example the comment in Lemma 3 on Page 18.

[2] `https://www.isa-afp.org/entries/Posix-Lexing.html`

death. There are well-known and extensive textbooks about regular expressions, for example [26, 31, 50, 51] to name just a few. Also the "academic field" appears to be extremely well-researched given the huge number of papers about regular expressions. Therefore it might be instructive to first have a look at why it makes sense to write a thesis about regular expressions in 2018?

One problem with regular expressions can be seen in the graph below: it plots the running times of the regular expression libraries built into Java 8 and current versions of Python when solving the problem whether the regular expression $(a^*)^*\,b$ matches strings of the form $\underbrace{a \ldots a}_{n}$.



The matching clearly always fails, but it is surprising that it takes such a long time to decide for even relatively small strings. After all there is classic work establishing that for a given regular expression such matching problems should be linear in the length of strings. This assumes the matching is done by using a DFA. But patently the graph above shows that in Java and Python the problem appears to be exponential. For example for the string consisting of just 28 a's, Python and Java need approximately 30 seconds to decide whether this string is matched or not, and for slightly longer strings one usually receives out-of-memory exceptions. While this particular regular expression and matching problem are slightly contrived, it is *not* the only instance where this happens. In fact, there are many more similarly simple regular expressions that show the same behaviour. There are also several other widely-used libraries, not just in Java and Python, behaving

in similar "exponential" manner.

That this is not just an "academic" problem is shown by reports where large software systems suddenly stopped working because of problems with regular expressions. For example, on 20 July 2016 a regular expression brought the popular webpage Stack Exchange to its knees.[3] The purpose of the regular expression was to trim unicode space from the start and end of lines, and a user post containing approximately 20,000 whitespaces in a comment line caused the server to go on high CPU loads such that the webpage became inaccessible. A similar problem was described in 2016 for a regular expression in the Atom editor.[4] There the purpose of the regular expression was to calculate the indentation of the next line. For one particular line a user had written, Atom needed to calculate for half an hour before writing a new line.[5] Another report from 2018 described a problem with a regular expression whose purpose is to match http-addresses. Again for a particular http-address the matching resulted in high CPU loads and exception traces, rather than the expected yes/no answer.

While the textbooks mentioned above do not feature anything about this phenomenon, it is somewhat well-known among engineers. Digging a bit deeper, it turns out the phenomenon has already been given a name—*catastrophic backtracking* [24]. There are also tools, called *regex debuggers*, which try to test when a regular expression is prone to catastrophic backtracking [1]. Usually such tests are sound, but not complete—meaning engineers cannot completely rely on them in order to recognise instances where catastrophic backtracking might occur. Digging even a bit further reveals that there is about a handful of research papers that take head on this issue. For example, Kirrage et al [29] use static analysis methods in order to detect potential instances of catastrophic

[3]The report by an engineer of Stack Exchange can be found at `http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016`.

[4]`https://atom.io`

[5]`http://davidvgalbraith.com/how-i-fixed-atom/`

backtracking. Other examples are Weideman et al [59] and Berglund et al [9]. The first two works also implement diagnostic tools for detecting potential instance of catastrophic backtracking, but do not offer direct help with rewriting the problematic regular expressions such that they are not susceptible any more to catastrophic backtracking, or with avoiding the problem altogether.

Catastrophic backtracking does not just happen with "negative" queries, that is when a regular expression does *not* match a string. It can also happen with "positive" queries. For example $(a^?)^{\{n\}}a^{\{n\}}$ is another candidate for catastrophic backtracking. In this regular expression the question mark stands for an *optional* match, that is match either a or the empty string, and $_^{\{n\}}$ stands for matching exactly n copies with n being a natural number. If one matches this regular expression against strings containing n a's, then many libraries behave in an exponential fashion. This is because they first attempt to match the a's using the optional part of the regular expression, and then need to backtrack in order to match with the exactly-n-times part of the regular expression. Since they have to backtrack over *all* choices, exponential runtime behaviour ensues. Unfortunately, it is not so easy to predict precisely which regular expression library behaves in which way and in which instance catastrophic backtracking occurs. For example Java (up to Version 8) and current versions of Python exhibit catastrophic backtracking with the example $(a^*)^*b$ and strings of the form $a \ldots a$. Also Ruby up to Version 2.2.0 struggled with this example, but in later versions the authors introduced an adhoc "optimisation" by rewriting $(a^*)^*$ to just a^*. While this optimisation seems harmless, it can actually open a can of worms: the trouble is that such an optimisation can easily affect the correctness involving submatches as pointed out by Kuklewicz [34]. Also regular expression matching in Java 9 got much faster in instances such as $(a^*)^*b$, but the running time is still much slower than the expected "linear behaviour".

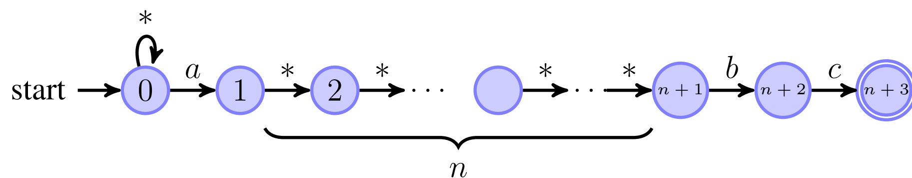
The root problem underlying catastrophic backtracking is that very many regular expression libraries, like in Java and Python, do not implement regular expression matchers based on DFAs, as one would expect, but based on NFAs. With NFAs the algorithm for reaching an accepting state needs to search and explore potentially alternative transitions. This search can be done in a *breadth-first fashion*. The problem with this method, however, is that it can result in rather high memory demands, as reported for example by Becchi and Crowley [7]. The idea is that when alternative transitions need to be explored one marks all candidate states as active and then recursively explores all transitions from those active states. The problem is that in typical matching problems almost all states become active resulting in a high memory bandwidth when the NFA contains thousands or even million states. This way of finding an accepting state is therefore not desirable in many applications. The alternative is to explore the candidate transitions in a *depth-first fashion*. Since this does not require much memory, it is often the implementation technique of choice for regular expression matching libraries. It is also often fast. . . just sometimes there are "outliers" which cause unexpected exponential behaviour and which can bring systems to a grinding halt, as mentioned above.

It is well-known that regular expressions can be translated into equivalent NFAs via the Thompson construction—this sometimes also called the McNaughton-Yamada-Thompson algorithm [38, 56]. NFAs can then be "determised" by the subset construction, and the resulting DFAs can be minimised via a myriad of minimisation algorithms. One interesting minimisation algorithm is by Brzozowski and has been formalised in Isabelle by Paulson [45]. This is a "well-rehearsed" approach to regular expression matching and well-explained in innumerable textbooks. The question therefore is why many existing regular expression libraries do not use DFAs where matching can be fast? There are two

answers: bounded repetitions and back-references.

Bounded Repetitions

One answer to the question is the slight, but significant, disconnect between what kind of regular expressions libraries need to support in practice, and what regular expressions are used in textbooks. One problematic kind of regular expression from practice are bounded repetitions, written $r^{\{n\}}$ with n being a natural number. The usual *un*bounded repetition in regular expressions is written with the Kleene-star as r^*. The point is that the bounded version requires that r needs to match exactly n times. Becchi and Crowley give an example that beautifully illustrates the problem with such bounded repetitions [7]: Consider the regular expression $.^*a\,.^{\{n\}}bc$ where the dots (each) match any character. Therefore the first part $.^*$ effectively means that the pattern prescribed by $a\,.^{\{n\}}bc$ can occur at any position of the input string. The pattern then searches for substrings starting with an a followed by n characters and ending with the characters bc. Becchi and Crowley give the following NFA for $.^*a\,.^{\{n\}}bc$



where the starred transitions can be performed for any input character. This is clearly a NFA, because in state 0 we do not know whether we should follow the transition to state 1 or remain in state 0 when receiving an a as input. Let us assume the input string is of the form $aaaaaaaaaaa\ldots aaaabc$. If we traverse the NFA in a breadth-first fashion, then state 0 will always be active, but also every a will make the transition $0 \rightarrow 1$ "fire" and this will activate state 1. So upon receiving a large enough stream of characters a's, all states from 0 to $n+1$ will

be active and need to be considered for potential transitions. Becchi and Crowley argue that this may result in unacceptable memory bandwidth requirements and too long processing times in practice.

Generating a DFA from the NFA above is also not really an option. The reason is that the subset construction, as is well-known, might blow up the size to 2^n states in the worst case. Unfortunately, NFAs involving bounded repetition will always hit this kind of blow-up because of the different values the "counter" n may take. It is still an active research area how to extend the traditional notion of automata (deterministic and non-deterministic) in order to deal more efficiently with such "boundedness constraints". The corresponding algorithms are deployed and relied upon in situations where processing times and memory demands are critical, but it appears to be unclear what the state of correctness claims and specifications are. Moreover, it is not clear what the relation of such automata is to POSIX matching/lexing (see later on).

Back-References

Another answer to why many existing regular expression libraries do not use DFAs has to do with *back-references*. They are part of *Perl Compatible Regular Expressions*, or PCRE for short.[6] Back-references are often indicated with numbers, such as $\backslash 1$, $\backslash 2$, ... Their idea is to match again a substring one has seen earlier in the input. To understand them better, assume you have the regular expression $a + b$, which can either recognise the character a or b. Then $(a + b)\backslash 1$ means we can recognise either a or b as first character, but then as second we want to have exactly the same character as again. So aa and bb would be OK, but ab or ba would not. Therefore the above regular expression is *not* equivalent to $(a + b)(a + b)$ where we just copy the relevant subexpression. The number in

[6] `http://www.pcre.org`

the back-reference refers to the corresponding "group" enclosed in parentheses. For example $(a + b)(c + d)\backslash 1\backslash 2$ accepts the strings

$$acac$$

$$bcbc$$

$$adad$$

$$bdbd$$

A possible application for back-references is recognising well-formed HTML-tags. These tags can be of the form $\backslash < tag >$ where tag could be anything like $head$, $body$ and so on. But then the requirement is that the closing tag should be the same tag again (just prefixed with a $\hat{}$). Back-references allow us to construct for this a regular expression of the form

$$\backslash < (tag) > \ldots \backslash <\hat{}\backslash 1 >$$

where the regular expression tag would prescribe which tags are to be matched and the back-reference $\backslash 1$ ensures that only strings with "matching" tags match successfully.

While adding bounded regular expressions to "normal" regular expressions is quite innocuous from a formal language point of view—it does not bring us outside the set of regular languages, adding back-references is a bit more serious—it allows us to recognise non-regular languages. Clearly back-references allow us to recognise "squares", such as $papa$ or $weewee$, using the PCRE $(.^{+})\backslash 1$. The point is that the language of squares is not regular and interestingly also not context-free.[7] Another such example is given by Câmpeanu et al [16] who prove that the language $\{a^n ba^n ba^n \mid n \geq 1\}$ is not context-free, but can be expressed as $(a^{+})b\backslash 1b\backslash 1$ by a PCRE. Furthermore, the language $\{a^n b^n \mid n \geq 0\}$ is context-free, but there is no regular expression, not even one involving back-references,

[7] See https://en.wikipedia.org/wiki/Regular_expression.

that could match it. They show that PCRE languages are properly contained in context-sensitive languages though.

While the non-regularity of back-references might be considered as an interesting "quirk", the real problem is that the resulting matching problem becomes NP-hard! This has been shown by a reduction to the k-vertex cover problem for graphs, which is known to be NP-hard. This reduction has been given by Aho [3] and is also nicely described by Rosulek.[8] (We shall therefore omit the details here). In light of this NP-hardness result, the choice of a depth-first search algorithm for regular expression matching, like implemented in Java and Python, does not look like such an absurd choice. If there were a more efficient algorithm to decide in general the matching problem involving back-references, we would also be able to quickly compute solutions for the graph 3-colourability problem etc, which is generally believed to be impossible.

There are a number of efficient regular expression libraries, for example Google's RE2, which are based on DFAs, but they do *not* support back-references. Why not ditching back-references then? Well, they seem to be useful to engineers: for example Snort contains around 8,000 regular expressions for monitoring network traffic and around 5 to 10% of them use back-references [7]. (These regular expressions are community curated and change from version to version depending on known attack patterns.)

Brzozowski Derivatives of Regular Expressions

This brings us to the main topic of this thesis: Brzozowski [15] introduced the notion of the *derivative*, written $r\backslash c$, of a regular expression r w.r.t. a character c, and showed that it gave 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

[8] See `http://www.mikero.com/misc/code/vertex-cover2.html`.

characters of the string matches the empty string, then r matches s (and *vice versa*). The derivative has the property (which may almost be regarded as its specification) that, for every string s and regular expression r and character c, one has $cs \in L(r)$ if and only if $s \in L(r\backslash c)$. Because of this attractive property, the central point in this thesis is to *not* use automata for regular expression matching, rather use Brzozowski's derivatives instead.

Brzozowski introduced derivatives of regular expressions in 1964. Since then they have acquired a somewhat interesting "history": Over the years derivatives of regular expressions were certainly known in the Formal Languages community and utilised for various purposes. An important research mile-stone, for example, is the notion of *partial derivatives* for regular expressions introduced by Antimirov in 1995 [5]. However, in the communities broadly described as Programming Languages and as Formal Proofs, they were largely forgotten. Owens et al wrote in 2009 that derivatives of regular expressions had been lost "*in the sands of time*" [43]. However, they recently have experienced a renaissance and become again a "hot" research topic with numerous research papers appearing in the last ten years—[4, 18, 19, 20, 55, 57, 60] to cite a few. Krishnaswami and Yallop even claim in a paper from 2018 that if somebody implements a regular expression matcher using derivatives, then "*you have almost surely identified a functional programmer*" [33].[9]

The beauty of Brzozowski's derivatives is that they are neatly expressible in any functional programming language—the code just consists of an algebraic datatype for regular expressions and two simple recursive functions. This simplicity and "algebraic nature" of derivatives is also the main attraction for theorem provers. The simple definitions can be easily rendered into theorem prover code and also very easily be reasoned about by performing inductions over in-

[9] https://www.cl.cam.ac.uk/~jdy22/papers/a-typed-algebraic-approach-to-parsing.pdf

ductive datatypes and recursive definitions. A consequence is that proving the correctness of the Brzozowski's matcher is a nice "afternoon exercise" in modern theorem provers. For example mechanised proofs can be found in HOL4, where a proof has been mentioned by Owens and Slind [44]. Another one can be found in Isabelle/HOL as part of the work by Krauss and Nipkow [32]. And another one in Coq is given by Coquand and Siles [20].

By using Brzozowski's derivatives for matching we also benefiting from the fact that regular expressions are more convenient for "composition"—be it sequential or alternative composition. The reason is that there are explicit constructors in regular expressions for composition. This allows us to reason compositionally about regular expressions—we can take them apart and put them together again. In contrast, a "formal" notion of composition in automata is not as straightforward and also is heavily sensitive to how automata are represented (possible representations are graphs, matrices, functions and so on). However, Paulson [45], and also Doczkal et al [22] take a somewhat opposing view and report that their formalisations of automata were rather "smooth". One advantage of automata, in comparison with regular expressions, however, is that there is a standard notion of what a minimal automata is, while there is no equivalent notion for regular expressions.

Another attraction of Brzozowski's derivatives is that they elegantly extend to additional constructors of regular expressions. Owens et al describe how the *not*-regular expression can be easily included in the definition of derivatives [43]. They also show that this regular expression is very convenient for prescribing patterns for recognising typical comments in programming languages, such as C-like comments of the form `/* ... */`. These patterns should start with a `/*`, but then the three dots should match anything *except* the final `*/`. Such constraints can be concisely expressed via the not-regular expression. Brzozowski's

derivatives also elegantly extend to bounded repetitions. While the details about the complexity are not yet fully worked out, it should be possible to treat bounded repetitions using Brzozowski derivatives *without* having to pay a heavy penalty in terms of processing time, in contrast to the penalty having to be paid by standard DFAs. Unfortunately, nothing is known yet about the relation of Brzozowski derivatives and back-references.

One limitation of Brzozowski's original derivative-based matcher is that it only generates a yes/no answer for whether a regular expression matches a string or not. Our motivation to look at this area arose from the paper by Sulzmann and Lu [53] which cleverly extends Brzozowski's matching algorithm to POSIX lexing. This extended version generates additional information on *how* a regular expression matches a string. We shall describe this in the next section.

1.2 POSIX Lexing

One application of regular expressions is in lexers. Lexers need to split up an input string into a sequence of tokens, each of which is frequently defined by a regular expression. Suppose r_{key} is a regular expressions for recognising key-words such as *if*, *then*, *while*, *for* and so on; and r_{id} a regular expression for identifiers—a single character followed by characters or numbers. The problem is that these regular expressions often "overlap", in the sense that a keyword usually also satisfies the constraints for an identifier. This problem can also occur within a single regular expression, because if a regular expression matches a string, then in general there is more than one way of how the string can be matched. There are two commonly used disambiguation strategies in order to generate a unique answer: one is called GREEDY matching [23] and the other is

POSIX matching [2, 34, 41, 53, 58].[10]

To see the difference between both strategies consider the string xy and the regular expression $(x + y + xy)^*$. Either the string can be matched in two 'iterations' by the single letter-regular expressions x and y, or directly in one iteration by xy. The first case corresponds to GREEDY matching, which first matches with the left-most symbol and only matches the next symbol in case of a mismatch (this is greedy in the sense of preferring instant gratification to delayed repletion). The second case is POSIX matching, which prefers the longest match. There are four informal rules behind tokenising a string in a POSIX fashion [2]:

- *The Longest Match Rule* (or "*maximal munch rule*"):

  The longest initial substring matched by any regular expression is taken as next token.

- *Rule Priority*:

  For a particular longest initial substring, the first regular expression that can match determines the token.

- *Star Rule*:

  A subexpression repeated by * shall not match an empty string unless this is the only match for the repetition.

- *Empty String Rule*:

  An empty string shall be considered to be longer than no match at all.

In the context of lexing, where an input string needs to be split up into a sequence of tokens, POSIX is the more natural disambiguation strategy for what programmers consider basic syntactic building blocks in their programs. Consider again

[10] POSIX matching acquired its name from the fact that the corresponding rules were described as part of the POSIX specification for Unix-like operating systems [2].

r_{key} for recognising keywords and r_{id} for recognising identifiers. Then we can form the regular expression $(r_{key} + r_{id})^*$ and use POSIX matching to tokenise strings, say *iffoo* and *if*. For *iffoo* we obtain by the Longest Match Rule a single identifier token, not a keyword followed by an identifier. For *if* we obtain by the Priority Rule a keyword token, not an identifier token—even if r_{id} matches also. By the Star Rule we know $(r_{key} + r_{id})^*$ matches *iffoo*, respectively *if*, in exactly one 'iteration' of the star. The Empty String Rule is for cases where, for example, the regular expression $(a^*)^*$ matches against the string bc. Then the longest initial matched substring is the empty string, which is matched by both the whole regular expression and the parenthesised subexpression.

While POSIX matching seems natural in a context of lexing, it turns out to be much more subtle than GREEDY matching in terms of implementations and in terms of proving properties about it. This was also noted by Kuklewicz [34] who found that nearly all POSIX matching implementations are "buggy" [53, Page 203] and by Grathwohl et al [25, Page 36] who 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."*

One should also not underestimate the difficulties when implementing POSIX matching using automata: Using a naive method, one has to follow transitions (according to the input string) until one finds an accepting state, record this state and look for further transitions which might lead to another accepting state that represents a longer initial substring. This might mean that one has to consider the entire string to make sure no other accepting state can be found. Only if none can be found, the last accepting state is returned. Yes, it can be done, but it takes quite some "head-standing" in order to get this process to run in linear time (see

for example [47]).

Given that POSIX matching is not so straightforward to implement and only informally defined by the rules in English shown above, Sulzmann and Lu correctly argued in [53] that this needs a formal specification. In order to establish the correctness of their algorithm, they define an "ordering relation" between *values* (possible outcomes for how a string can be matched by a regular expression) and argue that for every string and every regular expression, there is always a maximum value, as given by the their derivative-based algorithm.

The purpose of values is to encode the information of *how* a string is matched by the regular expression—that is, which part of the string is matched by which part of the regular expression. For this consider again the string xy and the regular expression $(x + (y + xy))^*$ (this time fully parenthesised). We can view this regular expression as a tree and if the string xy is matched by two Star 'iterations', then the x is matched by the left-most alternative in this tree and the y by the right-left alternative. This suggests to record this matching as

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

where Stars, Left, Right and Char are constructors for values. Stars records how many iterations were used; Left, respectively Right, which alternative is used. The value for the single 'iteration', i.e. the POSIX value, would look as follows

$$\mathit{Stars}\ [\mathit{Seq}\ (\mathit{Char}\ x)\ (\mathit{Char}\ y)]$$

where Stars has only a single-element list for the single iteration, and Seq indicates that xy is matched by a sequence regular expression. This 'tree view' leads naturally to the idea that regular expressions act as types and values as inhabiting those types (see, for example, [27] where this view is taken).

The approach of establishing that the matching algorithm generates a maximum value is inspired by work by Frisch and Cardelli [23] on GREEDY matching. In our formalisation effort, we made some partial attempts to formalise their specification of GREEDY matching and did *not* encounter any problems. This is in contrast with the work by Sulzmann and Lu where we hit almost immediately upon serious difficulties. While Sulzmann and Lu give a considerable amount of details for their correctness proof (some inside the paper and some more details in an appendix), this correctness proof is unformalised, meaning it is just a "pencil-and-paper" proof. In fact, we believe the purported proof they give does not work in central places. For example we were not able to establish the transitivity and totality properties for their "order relation" (the proofs of which were elided in [53]). We had some communication with Sulzmann about our problems via email. For example upon pointing out one problem uncovered by our formalisation, he commented:

> *"How could I miss this? Well, I was rather careless when stating this Lemma... Great example [of] how formal machine checked proofs (and proof assistants) can help to spot flawed reasoning steps."*

This led Sulzmann to augment some of the reasoning in their paper—he published an extended version of the paper on his website.[11] However, ultimately we abandoned the attempt to formalise Sulzmann and Lu's pencil-and-paper proof in Isabelle, because of the obstacles we encountered. In spite of this failure, we were eventually able to show the correctness of Sulzmann and Lu's matching algorithm by introducing our own notion for what a correct POSIX value is. This is an inductive definition inspired by work by Vansummeren [58]. Using this definition, the correctness of the algorithm can be established without too excessive formalisation work. We shall describe our formalisation next.

[11] `http://www.home.hs-karlsruhe.de/~suma0002/`

1.3 Preliminaries

In our Isabelle/HOL formalisation strings are lists of characters with the empty string being represented by the empty list, written [], and list-cons being written as $_ :: _$; string concatenation is $_ @ _$. Often we use the usual bracket notation for lists also for strings; for example a string consisting of just a single character c is written $[c]$. We also use the usual definitions for *prefixes* and *suffixes* of strings, as well as their *strict* versions. By using the type *char* for characters, we have a supply of finitely many characters roughly corresponding to the ASCII character set.

Regular Expressions are defined as an Isabelle/HOL inductive datatype. We start here with the standard textbook regular expressions with the following six constructors.

Definition 1. Regular expressions are given by the grammar:

$$
\begin{array}{rll}
r ::= & \mathbf{0} & \\
| & \mathbf{1} & \\
| & c & \text{single character} \\
| & r_1 + r_2 & \text{alternative / choice} \\
| & r_1 \cdot r_2 & \text{sequence} \\
| & r^* & \text{star (zero or more)}
\end{array}
$$

where $\mathbf{0}$ stands for the regular expression that does not match any string, $\mathbf{1}$ for the regular expression that matches only the empty string and c for matching a character literal (of type char). In what follows we shall sometimes omit the $\cdot$ in sequence regular expressions and just write $r_1\, r_2$ for brevity.

While the $\mathbf{0}$ does not play an essential role in works that use automata for regular expression matching, it is crucial for Brzozowski's derivatives. The regular

expressions defined above are often called *basic* regular expressions in order to distinguish them from *extended regular expressions* which may also include constructors for bounded repetitions, negation, optional regular expressions, and so on.

We next need some operations on *languages*, which are just sets of strings. We shall use the operation $_ @ _$ for the concatenation of two languages (it is also list-append for strings). The *Star* of a language, written $_\star$, is defined inductively by two clauses: *(i)* the empty string being in the star of a language and *(ii)* if s_1 is in a language and s_2 in the star of this language, then also $s_1 @ s_2$ is in the star of this language. We could also easily define the star of a language via the power operation as follows

$$A\star \stackrel{\text{def}}{=} \bigcup_{0 \leq n} A^n$$

Both definitions can be straightforwardly shown to be equivalent. Which definition to settle on is mainly a matter of taste: some later proofs can be found automatically in Isabelle using the former definition; other proofs are automatic with the latter one.

Later on it will also be convenient to use the following notion of a *semantic derivative* (or *left quotient*) of a language A with respect to a character c, defined as

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

This means in a semantic derivative we are looking for all strings in a set A starting with a character, say c, then strip off this character, and filtering out everything else. For semantic derivatives we have the following equations (for example mechanically proved in [32]):

$$
\begin{array}{lcl}
Der\, c\, \varnothing & = & \varnothing \\
Der\, c\, \{[]\} & = & \varnothing \\
Der\, c\, \{[d]\} & = & \textit{if } c = d \textit{ then } \{[]\} \textit{ else } \varnothing \\
Der\, c\, (A \cup B) & = & Der\, c\, A \;\cup\; Der\, c\, B \\
Der\, c\, (A \,@\, B) & = & (Der\, c\, A \,@\, B) \cup (\textit{if } [] \in A \textit{ then } Der\, c\, B \textit{ else } \varnothing) \\
Der\, c\, (A\star) & = & Der\, c\, A \,@\, A\star
\end{array}
\tag{1.1}
$$

The main definition for regular expressions is the *associated language*, written $L(_)$, and defined recursively as follows.

Definition 2. The *associated language* of r, written $L(r)$, is defined as follows:

$$
\begin{array}{rcl}
L(\mathbf{0}) & \stackrel{\text{def}}{=} & \varnothing \\
L(\mathbf{1}) & \stackrel{\text{def}}{=} & \{[]\} \\
L(c) & \stackrel{\text{def}}{=} & \{[c]\} \\
L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) \\
L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L(r_1) \,@\, L(r_2) \\
L(r^*) & \stackrel{\text{def}}{=} & (L(r))\star
\end{array}
$$

The main point of the L-function is that we can use it to precisely specify when a string s is matched by a regular expression r, namely if and only if $s \in L(r)$. This is clearly a specification because in the star-clause, the language can be infinite and a membership test for an infinite set cannot be directly implemented. Below we shall use the terminology that a regular expression r "matches the language $L(r)$", that is matches every string in $L(r)$. We can also use L to define the equivalence of two regular expressions, which will be needed when we need to simplify regular expressions.

Definition 3. Two regular expressions are *equivalent* iff they match the same language, namely

$$r_1 \equiv r_2 \stackrel{\text{def}}{=} L(r_1) = L(r_2)$$

Central to Brzozowski's regular expression matcher are two functions called $\mathit{nullable}$ and *derivative*. The latter is written $r\backslash c$ for the derivative of the regular expression r w.r.t. the character c. Both functions are defined by recursion over regular expressions.

Definition 4.

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

The derivative function takes a regular expression, say r and a character, say c, as input and returns the derivative regular expression.

Definition 5.

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

The idea behind $nullable$ is relatively clear: it tests whether a regular expression can match the empty string. By contrast, the idea behind the derivative might be less clear. To see what is going on, suppose a regular expression r can match strings of the form $c :: s$, then the derivative function answers the question what does the regular expression look like that can match the string s (where the leading character c has been "chopped off")? Let us see how this characterisation is reflected in the clauses of the derivative function.

The first two clauses of the derivative are straightforward: for this recall that $r\backslash c$ should calculate a regular expression so that given the "input" regular expression can match a string of the form $c :: s$, we want a regular expression for s. Since neither **0** nor **1** can match a string of the form $c :: s$, we return **0**. In the character clause we have to make a case-distinction: In case the regular expression is c, then clearly it can recognise a string of the form $c :: s$, just that s is the empty string. Therefore we return the **1**-regular expression. In the other case we again return **0** since no string of the form $c :: s$ can be matched.

Elucidating the recursive clauses is a bit more involved. Fortunately, the $+$-case is still relatively simple: all strings of the form $c :: s$ are either matched by the regular expression r_1 or r_2. So we just have to recursively call the derivative with these two regular expressions and compose the results again with $+$. The $\cdot$-clause is more complicated: if $r_1 \cdot r_2$ matches a string of the form $c :: s$, then the first part must be matched by r_1. Consequently, it makes sense to construct the regular expression for s by calling the derivative with r_1 and "appending" r_2. There is however one exception to this simple rule: if r_1 can match the empty string, then all of $c :: s$ can be matched by r_2. Consequently in case r_1 is $nullable$ (that is can match the empty string) we have to allow the choice $r_2\backslash c$ for calculating the regular expression that can match s. This means we have to add the regular expression $r_2\backslash c$ in the result. The *-clause is again simple: if r^*

matches a string of the form $c :: s$, then the first part must be "matched" by a single copy of r. Therefore we call recursively $r\backslash c$ and "append" r^* in order to match the rest of s.

We can extend the derivative of regular expressions from single characters to strings as follows:

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

Before we go on, let us look at an example. Suppose the regular expression r_0 is $(a + ab) \cdot (b + \mathbf{1})$ and the input string is ab. Clearly r_0 can match the input string. In fact there are two ways for how it can match this string. Below we give the intermediate steps for calculating the derivative $r_0\backslash[a, b]$:

$$r_1 = r_0\backslash a: \quad (\mathbf{1} + \mathbf{1}\, b) \cdot (b + \mathbf{1})$$

One can see that this derivative can match the string $[b]$, again in two ways. Next we have

$$r_2 = r_1\backslash b: \quad (\mathbf{0} + \mathbf{0}\, b + \mathbf{1}) \cdot (b + \mathbf{1}) + (\mathbf{1} + \mathbf{0}) \tag{1.2}$$

The point of the last derivative is that we can decide whether it matches the empty string: in this case it does and again in two ways. Given the idea behind the derivative operation, it is relatively easy to convince oneself of the fact that if the last derivative matches the empty string, then the original regular expression matches the string that was used for building the derivative. This holds also in the other direction.

Using *nullable* and the derivative operation, we can define the following simple regular expression matcher:

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

This is essentially Brzozowski's algorithm from 1964. Its main virtue is that the algorithm can be easily implemented as a functional program (either in a functional programming language or in a theorem prover). The correctness proof for $match$ amounts to establishing the property

$$match\ s\ r \text{ if and only if } s \in L(r) \tag{1.3}$$

On the left-hand side of this property is the algorithm; on the right-hand side its specification. For this proof to go through, we need the following two auxiliary properties.

Lemma 1.

(1) $nullable(r)$ *if and only if* $[] \in L(r)$

(2) $L(r\backslash c) = Der\ c\ L(r)$

Proof. The first is by a simple induction on r. Given the equations in (1.1) the second is also by a simple induction on r. □

We can then prove (1.3) by an induction on s generalising over r and using the above two properties. The ease of the these proofs is the main attraction for theorem provers—it is a nice formalisation exercise, for example done by Owens and Slind [44] using the HOL theorem prover, but is also part of the *Archive of Formal Proofs* for Isabelle. The novel idea of Sulzmann and Lu is to append another phase to Brzozowski's algorithm in order to calculate a (POSIX) value. We will explain this next.

1.4 POSIX Lexing Algorithm by Sulzmann and Lu

Sulzmann and Lu presented their POSIX lexing algorithm in 2014 [53]. This algorithm consists of two phases: first a matching phase (which is Brzozowski's

algorithm) and then a value construction phase. The values encode *how* a regular expression matches a string. The grammars below show regular expressions together with their corresponding values:

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

As can be seen for each regular expression there is a specific value that records how the regular expression matched the string. For example $\mathit{Char}(c)$ is the value for the character regular expression c. Similarly Seq for the sequence regular expression. The exception is the **0**-regular expression, because it cannot match anything and therefore does not need a corresponding value; and also the two values, Left and Right, for the alternative regular expression, which correspond to the two choices in the alternative. So if we are given a value, it will always be clear what the corresponding (kind) of regular expression is—whether it is a sequence regular expression and so on. This holds also in the other direction: if we are given a regular expression, it will be clear what the form of the corresponding value must be.

We sometimes need to extract the string "underlying" a value. This can be done with the *flatten* function written $|_|$:

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

We will often refer to the underlying string of a value as the *flattened value*. We will also overload our notation and use $|vs|$ for flattening a list of values and concatenating the resulting strings.

Sulzmann and Lu [53] define inductively a kind of *type inhabitation relation* that associates values to regular expressions. We define this relation as follows:[12]

Definition 6 (Inhabitation Relation)**.**

$$
\frac{}{\vdash Empty : \mathbf{1}} \qquad \frac{}{\vdash Char\ c : c}
$$

$$
\frac{\vdash v_1 : r_1}{\vdash Left\ v_1 : r_1 + r_2} \qquad \frac{\vdash v_2 : r_2}{\vdash Right\ v_2 : r_1 + r_2}
$$

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

In the clause for $Stars$ we use the notation $v \in vs$ for indicating that v is a member in the list vs. We require in this rule that every value in vs flattens to a non-empty string. The idea is that $Stars$-values satisfy the informal Star Rule

[12]Note that the rule for *Stars* differs from our conference paper [6]. There we used the original definition by Sulzmann and Lu [53] which does not require that the values $v \in vs$ flatten to a non-empty string. The reason for introducing the more restricted version of lexical values is convenience later on when reasoning about an ordering relation for values.

from POSIX (see Section 1.2) where the * does not match the empty string unless this is the only match for the repetition. Note also that no values are associated with the regular expression **0**, and that the only value associated with the regular expression **1** is $Empty$. It is routine to establish how values "inhabiting" a regular expression correspond to the language of a regular expression, namely

Proposition 1. $L(r) = \{|v| \mid \vdash v : r\}$

Given a regular expression r and a string s, we define next the set of all *Lexical Values*, written $LV\ r\ s$, inhabited by r with the underlying string being s.[13]

Definition 7 (Lexical Values)**.**

$$LV\ r\ s \stackrel{\text{def}}{=} \{v \mid \vdash v : r \wedge |v| = s\}$$

The main property of the set $LV\ r\ s$ is that it is always finite.

Lemma 2. For all r and s, *finite* $(LV\ r\ s)$.

Proof. By induction on r generalising over s. The only interesting cases are $r_1 \cdot r_2$ and r^*. In the first case we reason as follows: $LV\ (r_1 \cdot r_2)\ s$ is a subset of $\{Seq\ v_1\ v_2 \mid v_1 \in Pre \wedge v_2 \in Suf\}$ where Pre and Suf are defined as follows:

$$\begin{aligned} Pre &\stackrel{\text{def}}{=} \bigcup_{s' \in Prefixes\ s} .LV\ r_1\ s' \\ Suf &\stackrel{\text{def}}{=} \bigcup_{s' \in Suffixes\ s} .LV\ r_2\ s' \end{aligned}$$

Since for a given string s, there are only finitely many prefixes and suffixes, we know by induction hypothesis that Pre and Suf are finite sets of values. So also $LV\ (r_1 \cdot r_2)\ s$ must be finite. In case of $LV\ (r^*)\ s$ we reason similarly, except

[13] Okui and Suzuki refer to our lexical values as *canonical values* in [41]. The notion of *non-problematic values* by Cardelli and Frisch [23] is related, but not identical to our lexical values.

that this set is a subset of $\{Stars\,[]\} \cup \{Stars\,(v :: vs) \mid v \in Pre \wedge vs \in SSuf\}$ where $SSuf$ is the set of lexical values built from the strict suffixes (suffixes that are shorter than s). It is sufficient to only consider strict suffixes, because of the side-condition about values not flattening to the empty string. □

This finiteness property does not hold in general if we remove the side-condition about $|v| \neq []$ in the $Stars$-rule above. For example using Sulzmann and Lu's less restrictive definition, $LV\,(\mathbf{1}^*)\,[]$ would contain infinitely many values, but according to our more restricted definition only a single value, namely $LV\,(\mathbf{1}^*)\,[] = \{Stars\,[]\}$. This more restricted version of lexical values will be useful later on when we show that our POSIX specification is equivalent to the one by Okui and Suzuki.

If a regular expression r matches a string s, then generally the set $LV\ r\ s$ is not just a singleton set. In case of POSIX matching the problem is to calculate the unique lexical value that satisfies the (informal) POSIX rules from Section 1.2. Sulzmann and Lu give such an algorithm. Graphically their POSIX value calculation algorithm can be illustrated by the picture in Figure 1.1 where the path from the left to the right involving derivatives/$nullable$ is the first phase of the algorithm (calculating successive Brozowski's derivatives) and $mkeps$/inj, the path from right to left, the second phase. This picture shows the steps required when a regular expression, say r_1, matches the string $[a, b, c]$. We first build the three derivatives (according to a, b and c). We then use $nullable$ to find out whether the resulting derivative regular expression r_4 can match the empty string. If yes, we call the function $mkeps$ that produces a value v_4 for how r_4 can match the empty string (taking into account the POSIX constraints in case there are several ways—we shall explain this below). Then we call the injection function to inject "back" the letters c, b and a in order to obtain the value v_1 that encodes how r_1 matches the string abc.

The function $mkeps$ is defined as follows:

Definition 8.

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

Notice how this function makes some subtle choices leading to a POSIX value: for example if the alternative, say $r_1 + r_2$, can match the empty string and furthermore r_1 can match the empty string, then we return always a $\textit{Left}$-value. The $Right$-value will only be returned if r_1 is *not* $nullable$. The four regular expressions in $mkeps$ are the only cases we need to consider, since the other regular expressions cannot match the empty string. Recall the derivative r_2 from (1.2):

$$(\mathbf{0} + (\mathbf{0}\,b + \mathbf{1})) \cdot (b + \mathbf{1}) + (\mathbf{1} + \mathbf{0})$$

Below is the calculation of $mkeps$ including the intermediate steps.

$$
\begin{array}{ll}
 & mkeps((\mathbf{0} + (\mathbf{0}\,b + \mathbf{1})) \cdot (b + \mathbf{1}) + (\mathbf{1} + \mathbf{0})) \\
= & \textit{Left}(mkeps((\mathbf{0} + (\mathbf{0}\,b + \mathbf{1})) \cdot (b + \mathbf{1}))) \\
= & \textit{Left}(\textit{Seq}(mkeps(\mathbf{0} + (\mathbf{0}\,b + \mathbf{1})), mkeps(b + \mathbf{1}))) \\
= & \textit{Left}(\textit{Seq}(Right(mkeps(\mathbf{0}\,b + \mathbf{1})), Right(mkeps(\mathbf{1})))) \\
= & \textit{Left}(\textit{Seq}(Right(Right(mkeps(\mathbf{1}))), Right(\textit{Empty}))) \\
= & \textit{Left}(\textit{Seq}(Right(Right(\textit{Empty})), Right(\textit{Empty})))
\end{array}
$$

This means the value calculated by $mkeps$ corresponds to the two underlined $\mathbf{1}$s which in r_2 are responsible, according to the POSIX rules, for matching the empty string.

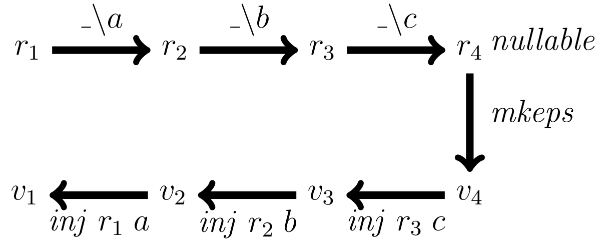


Figure 1.1: The two phases of the algorithm by Sulzmann & Lu [53], matching the string $[a, b, c]$. The first phase (the arrows from left to right) is Brzozowski's matcher building successive derivatives. If the last regular expression is $nullable$, then the functions of the second phase are called (the top-down and right-to-left arrows): first $mkeps$ calculates a value v_4 witnessing how the empty string has been recognised by v_4. After that the function inj "injects back" the characters of the string into the values.

$$(\mathbf{0} + (\mathbf{0}\, b + \underline{\mathbf{1}})) \cdot (b + \underline{\mathbf{1}}) + (\mathbf{1} + \mathbf{0})$$

The function $mkeps$ does not choose the right-most **1**, which would match the empty string as well, because this would violate the Priority Rule.

The really interesting function Sulzmann and Lu introduced in the second phase is called *injection* and written inj. Remember that the derivative essentially "chops off" a single character from a regular expression. The injection function undoes this "chopping off" by injecting back a character…just on the level of values, rather than regular expressions.

Definition 9. The inj function takes a regular expression, a character and a value as arguments; it produces another value. It is defined recursively as follows:

$$
\begin{array}{lll}
(1) & inj\ d\ c\ (Empty) & \stackrel{\text{def}}{=} & Char\ c \\
(2) & inj\ (r_1 + r_2)\ c\ (Left\ v_1) & \stackrel{\text{def}}{=} & Left\ (inj\ r_1\ c\ v_1) \\
(3) & inj\ (r_1 + r_2)\ c\ (Right\ v_2) & \stackrel{\text{def}}{=} & Right\ (inj\ r_2\ c\ v_2) \\
(4) & inj\ (r_1 \cdot r_2)\ c\ (Seq\ v_1\ v_2) & \stackrel{\text{def}}{=} & Seq\ (inj\ r_1\ c\ v_1)\ v_2 \\
(5) & inj\ (r_1 \cdot r_2)\ c\ (Left\ (Seq\ v_1\ v_2)) & \stackrel{\text{def}}{=} & Seq\ (inj\ r_1\ c\ v_1)\ v_2
\end{array}
$$

$$\begin{array}{llcl}
(6) & inj\ (r_1 \cdot r_2)\ c\ (Right\ v_2) & \stackrel{\text{def}}{=} & Seq\ (mkeps\ r_1)\ (inj\ r_2\ c\ v_2) \\
(7) & inj\ (r^*)\ c\ (Seq\ v\ (Stars\ vs)) & \stackrel{\text{def}}{=} & Stars\ (inj\ r\ c\ v :: vs)
\end{array}$$

To better understand what is going on in this definition it might be instructive to look first at the three sequence cases (clauses *(4) – (6)*). In each case we need to construct an "injected value" for $r_1 \cdot r_2$. This must be a value of the form $Seq\ _\ _$. Recall the clause of the derivative-function for sequence regular expressions:

$$(r_1 \cdot r_2) \backslash c \stackrel{\text{def}}{=} \textit{if nullable } r_1 \textit{ then } (r_1 \backslash c) \cdot r_2 + (r_2 \backslash c) \textit{ else } (r_1 \backslash c) \cdot r_2$$

Consider first the *else*-branch where the derivative is $(r_1 \backslash c) \cdot r_2$. The corresponding value must therefore be of the form $Seq\ v_1\ v_2$, which matches the left-hand side in clause *(4)* of inj. In the *if*-branch the derivative is an alternative, namely $(r_1 \backslash c) \cdot r_2 + (r_2 \backslash c)$. This means we either have to consider a $Left$- or $Right$-value. In case of the $Left$-value we know further it must be a value for a sequence regular expression. Therefore the pattern we match in the clause *(5)* is $Left\ (Seq\ v_1\ v_2)$, while in *(6)* it is just $Right\ v_2$. One more interesting point is in the right-hand side of clause *(6)*: since in this case the regular expression r_1 does not "contribute" to matching the string, that means it only matches the empty string, we need to call $mkeps$ in order to construct a value for how r_1 can match this empty string. A similar argument applies for why we can expect in the left-hand side of clause *(7)* that the value is of the form $Seq\ v\ (Stars\ vs)$—the derivative of a star is $(r \backslash c) \cdot r^*$. Finally, the reason for why we can ignore the second argument in clause *(1)* of inj is that it will only ever be called in cases where $c = d$, but the usual linearity restrictions in patterns do not allow us to build this constraint explicitly into our function definition.[14]

[14]Sulzmann and Lu state this clause as $inj\ c\ c\ (Empty) \stackrel{\text{def}}{=} Char\ c$, but our deviation is harmless.

The idea of the inj-function to "inject" a character, say c, into a value can be made precise by the first part of the following lemma: It shows that the underlying string of an injected value has a prepended character c; the second part shows that the underlying string of an $mkeps$-value is always the empty string (given the regular expression is nullable since otherwise $mkeps$ might not be defined).

Lemma 3.

(1) *If* $\vdash v : r$ *then* $|inj\ r\ c\ v| = c :: |v|$.

(2) *If* $nullable(r)$ *then* $|mkeps\ r| = []$.

Proof. Both properties are by routine inductions: the first one can, for example, be proved by induction over the definition of derivatives; the second by an induction on r. There are no interesting cases. □

Recall the value from the above calculation, which was the result of $mkeps$

$$v = Left(Seq(Right(Right(Empty)), Right(Empty)))$$

and the derivative r_1 which is the derivative just before the last one in our derivative calculation

$$r_1 = (\mathbf{1} + \mathbf{1}\, b) \cdot (b + \mathbf{1})$$

Below are the intermediate steps for calling the inj function with r_1, b and v:

$$\begin{aligned}
& inj\ ((\mathbf{1} + \mathbf{1}b) \cdot (b + \mathbf{1}))\ \ b \\
& \qquad\qquad (Left(Seq(Right(Right(Empty)), Right(Empty)))) \\
& = Seq(inj\ (\mathbf{1} + \mathbf{1}b)\ b\ Right(Right(Empty)), Right(Empty)) \\
& = Seq(Right(inj(\mathbf{1}b)\ b\ Right(Empty)), Right(Empty)) \\
& = Seq(Right(Seq(mkeps(\mathbf{1}), inj\ b\, b\ Empty)), Right(Empty)) \\
& = Seq(Right(Seq(Empty, Char(b))), Right(Empty))
\end{aligned}$$

While the flattened value of v is the empty string, flattening the result of injecting b into this value (using r_1) gives use the string $[b]$, as expected. If we further inject back a into this value using r_0, which is

$$r_0 = (a + \underline{ab}) \cdot (b + \underline{\mathbf{1}})$$

we obtain

$$\mathit{Seq}(\mathit{Right}(\mathit{Seq}(\mathit{Char}(a), \mathit{Char}(b))), \mathit{Right}(\mathit{Empty}))$$

This value corresponds to underlined parts in r_0 (see above) and its flattened value is $[a, b]$.

Having defined the mkeps and inj function, we can extend Brzozowski's matcher so that a value is constructed (assuming the regular expression matches the string). The two clauses of the Sulzmann and Lu lexer are

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

We call this a *lexer*, because it produces a value that encodes how a regular expression matched a string, as opposed to a *matcher* which just produces a yes/no answer. In the lexer above, if the regular expression does not match the string, *None* is returned. If the regular expression *does* match the string, then *Some* value is returned. Like Sulzmann and Lu, we like to prove that this value is a POSIX value. We shall do this in the next chapter by using our own definition of what a POSIX value should be.

Chapter 2

Specification of POSIX Values

In this chapter, we will give our own inductive definition for POSIX values and show that the lexer discussed in the previous chapter always generates POSIX values. Our definition is inspired by work by Vansummeren [58]. We were unable to use the original definition by Sulzmann and Lu, because we could not establish some central properties for this definition. To give more confidence that our own definition captures the "spirit" of POSIX, we also show that it is in fact equivalent to a definition given by Okui and Suzuki [41] (their definition uses a different technique). Next, we shall discuss the GREEDY ordering rules by Firsch and Cardelli [23] which Sulzmann and Lu cite as the place where they have taken their main idea from for their correctness proof. We shall subsequently argue why the correctness proof by Sulzmann and Lu contains unfillable gaps.

2.1 Our POSIX Definition

Recall the informal POSIX rules described in Section 1.2. We shall formalise them in an inductive definition for a ternary relation. Our definition is inspired

$$\frac{}{([], \mathbf{1}) \rightarrow \mathit{Empty}}P1 \qquad \frac{}{([c],\ c) \rightarrow \mathit{Char}\ c}PC$$

$$\frac{(s,\ r_1) \rightarrow v}{(s,\ r_1 + r_2) \rightarrow \mathit{Left}\ v}P+L \qquad \frac{(s,\ r_1) \rightarrow v \quad s \notin L(r_1)}{(s,\ r_1 + r_2) \rightarrow \mathit{Right}\ v}P+R$$

$$\frac{\begin{array}{c}(s_1,\ r_1) \rightarrow v_1 \quad (s_2,\ r_2) \rightarrow v_2 \\ \nexists s_3\ s_4.\ s_3 \neq []\ \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L(r_1) \wedge s_4 \in L(r_2)\end{array}}{(s_1 @ s_2,\ r_1 \cdot r_2) \rightarrow \mathit{Seq}\ v_1\ v_2}PS$$

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

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

Figure 2.1: Our inductive definition of POSIX values.

by the matching relation given by Vansummeren [58], where the corresponding rules incorporate the POSIX-specific choices about which value to prefer into the side-conditions (some of the side-conditions might need further explanation, which we shall give later). Our POSIX relation is written $(s, r) \rightarrow v$ relating strings, regular expressions and (POSIX) values. The corresponding inductive rules are given in Figure 2.1.

We can prove that given a string s and a regular expression r, the value v is uniquely determined by $(s, r) \rightarrow v$. Therefore we use the suggestive notation using an arrow (for "yields") in our POSIX specification.

Theorem 1.

(1) If $(s, r) \rightarrow v$ then $s \in L(r)$ and $|v| = s$.

(2) If $(s, r) \rightarrow v$ and $(s, r) \rightarrow v'$ then $v = v'$.

Proof. Both are by induction on the definition of $(s, r) \rightarrow v$. The second parts follows by a case analysis of $(s, r) \rightarrow v'$ and the first part. □

We claim that $(s, r) \rightarrow v$ captures the idea behind the four informal POSIX rules

described in Section 1.2. Consider for example the rules $P + L$ and $P + R$ where the POSIX value for a string and an alternative regular expression, that is $(s,\ r_1 + r_2)$, is specified—it is always a $Left$-value, *except* when the string to be matched is not in the language of r_1; only then it is a $Right$-value (see the side-condition in $P + R$). Interesting is also the rule for sequence regular expressions (PS). The first two premises state that v_1 and v_2 are the POSIX values for (s_1, r_1) and (s_2, r_2) respectively. Consider now the third premise and note that the POSIX value of this rule should match the string $s_1 @ s_2$. According to the Longest Match Rule, we want that the s_1 is the longest initial split of $s_1 @ s_2$ such that s_2 is still recognised by r_2. Let us assume, contrary to the third premise, that there *exist* an s_3 and s_4 such that s_2 can be split up into a non-empty string s_3 and a possibly empty string s_4. Moreover the longer string $s_1 @ s_3$ can be matched by r_1 and the shorter s_4 can still be matched by r_2. In this case s_1 would *not* be the longest initial split of $s_1 @ s_2$ and therefore $Seq\, v_1\, v_2$ cannot be a POSIX value for $(s_1 @ s_2, r_1 \cdot r_2)$. The main point is that our side-condition ensures the Longest Match Rule is satisfied.

A similar condition is imposed on the POSIX value in the $P*$-rule. Also there we want that s_1 is the longest initial split of $s_1 @ s_2$ and furthermore the corresponding value v cannot be flattened to the empty string. In effect, we require that in each "iteration" of the star, some non-empty substring needs to be "chipped" away; only in case of the empty string we accept $Stars\, []$ as the POSIX value. Indeed we can show that our POSIX values are lexical values which exclude those *Stars* that contain "impropper" subvalues that flatten to the empty string.

Lemma 4. *If* $(s, r) \rightarrow v$ *then* $v \in LV\ r\ s$.

Proof. By a straightforward induction on the POSIX definition. □

Next is the lemma that shows the function $mkeps$ by Sulzmann and Lu calculates the POSIX value for the empty string and a nullable regular expression.

Lemma 5. *If nullable* r *then* $([],\ r) \rightarrow mkeps(r)$.

Proof. By routine induction on r. ☐

The central lemma for our POSIX relation is that the inj-function preserves POSIX values.

Lemma 6. *If* $(s,\ r\backslash c) \rightarrow v$ *then* $(c :: s,\ r) \rightarrow$ *inj* $r\ c\ v$.

Proof. By induction on r. We explain two cases:

- Case $r = r_1 + r_2$. There are two subcases, namely (a) $v = \mathit{Left}\, v'$ and $(s,\ r_1\backslash c) \rightarrow v'$; and (b) $v = \mathit{Right}\, v',\ s \notin L(r_1\backslash c)$ and $(s,\ r_2\backslash c) \rightarrow v'$. In (a) we know $(s,\ r_1\backslash c) \rightarrow v'$, from which we can infer $(c :: s,\ r_1) \rightarrow \mathit{inj}\, r_1\ c\ v'$ by induction hypothesis and hence $(c :: s,\ r_1 + r_2) \rightarrow \mathit{inj}\ (r_1 + r_2)\ c\ (\mathit{Left}\, v')$ as needed. Similarly in subcase (b) where, however, in addition we have to use Proposition 1(2) in order to infer $c :: s \notin L(r_1)$ from $s \notin L(r_1\backslash c)$.

- Case $r = r_1 \cdot r_2$ and $s = s_1 \mathbin{@} s_2$. There are three subcases:

  (a) $v =$ *Left* (*Seq* $v_1\ v_2$) and *nullable* r_1

  (b) $v =$ *Right* v_1 and *nullable* r_1

  (c) $v =$ *Seq* $v_1\ v_2$ and $\neg$ *nullable* r_1

  For (a) we know $(s_1,\ r_1\backslash c) \rightarrow v_1$ and $(s_2,\ r_2) \rightarrow v_2$ as well as

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

From the latter we can infer by Proposition 1(2):

$$\nexists\, s_3\ s_4 \cdot s_3 \neq []\ \wedge s_3 @ s_4 = s_2 \wedge c :: s_1 @ s_3 \in L(r_1) \wedge s_4 \in L(r_2)$$

We can use the induction hypothesis for r_1 to obtain $(c\ ::\ s_1,\ r_1) \rightarrow$ $\mathit{inj}\, r_1\ c\ v_1$. Putting this all together allows us to infer $(c\ ::\ s_1 \mathbin{@} s_2,\ r_1 \cdot r_2) \rightarrow \mathit{Seq}\,(\mathit{inj}\, r_1\ c\ v_1)\ v_2$. The case (c) is similar.

For (b) we know $(s,\ r_2\backslash c) \rightarrow v_1$ and $s_1 @ s_2 \notin L((r_1\backslash c) \cdot r_2)$. From the former we have $(c :: s,\ r_2) \rightarrow \mathit{inj}\, r_2\ c\ v_1$ by induction hypothesis for r_2. From the latter we can infer

$$\nexists\, s_3\ s_4 \cdot s_3 \neq []\ \wedge s_3 @ s_4 = c :: s \wedge s_3 \in L(r_1) \wedge s_4 \in L(r_2)$$

By Lemma 5 we know $([], r_1) \rightarrow \mathit{mkeps}\, r_1$ holds. Putting this all together, we can conclude with

$$(c :: s,\ r_1 \cdot r_2) \rightarrow \mathit{Seq}\,(\mathit{mkeps}\, r_1)\,(\mathit{inj}\, r_2\ c\ v_1)$$

as required.

- Case $r = r_1^*$. This case is very similar to the sequence case, except that we need to also ensure that $|\mathit{inj}\, r_1\ c\ v_1| \neq []$. This follows from Lemma 3 and $(c\ ::\ s_1,\ r_1) \rightarrow \mathit{inj}\, r_1\ c\ v_1$ (which in turn follows from $(s_1,\ r_1\backslash c) \rightarrow v_1$ and the induction hypothesis).

□

With Lemma 6 in place, it is completely routine to establish that the Sulzmann and Lu lexer satisfies our specification (returning the null value *None* iff the string is not in the language of the regular expression, and returning a POSIX value iff the string *is* in the language):

Theorem 2.

(1) $s \notin L(r)$ *if and only if* *lexer* $r\ s =$ *None*

(2) $s \in L(r)$ *if and only if* $\exists\, v.$ *lexer* $r\ s =$ *Some* $v \wedge (s,\ r) \to v$

Proof. By induction on s using Lemma 5 and 6. □

In *(2)* we further know by Theorem 1 that the value returned by the lexer must be unique. A simple corollary of our two theorems therefore is:

Corollary 1.

(1) *lexer* $r\ s =$ *None if and only if* $\nexists\, v.\ (s,\ r) \to v$

(2) *lexer* $r\ s =$ *Some* v *if and only if* $(s,\ r) \to v$

This concludes our correctness proof. Note that we have not changed the algorithm of Sulzmann and Lu,[1] but introduced our own specification for what a correct result—a POSIX value—should be. Unfortunately, even with this result in place, we were unable to make any progress with formalising the original proof approach by Sulzmann and Lu. In the next section we shall show that our POSIX specification coincides with another one given by Okui and Suzuki using a different technique. This gives us more confidence that our definition really captures the idea behind the POSIX rules.

2.2 Ordering of Values According to Okui and Suzuki

While in the previous section we have defined POSIX values directly in terms of a ternary relation (see inference rules in Figure 2.1), Sulzmann and Lu took a

[1] All deviations we introduced are harmless.

different approach in [53]: they introduced an ordering for values and identified POSIX values as the maximal elements. A somewhat similar ordering was introduced by Okui and Suzuki [41, 42], which they use to establish the correctness of their automata-based algorithm for POSIX matching. Their ordering resembles some aspects of the one given by Sulzmann and Lu, but overall is quite different. To begin with, Okui and Suzuki identify POSIX values as minimal, rather than maximal, elements in their ordering. A more substantial difference is that the ordering by Okui and Suzuki uses *positions* in order to identify and compare subvalues.

Positions are lists of natural numbers. They allow one to easily identify subtrees by following individual branches in each level of trees. This is a well-known technique, for example in term rewriting [11], in order to identify subterms in larger terms. The position technique allows Okui and Suzuki to quite naturally formalise the Longest Match and Priority rules of the informal POSIX standard. Consider for example a value v of the form

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

At position $[0]$ of this value is the subvalue $Seq\ (Char\ x)\ (Char\ y)$; at position $[0,\ 1]$ is $Char\ y$ and at position $[1]$ the subvalue $Char\ z$. At the 'root' position, or empty list $[]$, is the whole value v. Positions such as $[0, 1, 0]$ or $[2]$ are outside of v. If it exists, the subvalue of v at a position p, written $v\downharpoonleft_p$, can be recursively defined by

$$\begin{array}{rcl}
v\downharpoonleft_{[]} & \stackrel{\text{def}}{=} & v \\
\textit{Left}\ v\downharpoonleft_{0::ps} & \stackrel{\text{def}}{=} & v\downharpoonleft_{ps} \\
\textit{Right}\ v\downharpoonleft_{1::ps} & \stackrel{\text{def}}{=} & v\downharpoonleft_{ps} \\
\textit{Seq}\ v_1\ v_2\downharpoonleft_{0::ps} & \stackrel{\text{def}}{=} & v_1\downharpoonleft_{ps}
\end{array}$$

$$\begin{array}{rcl} \textit{Seq}\ v_1\ v_2\downharpoonleft_{1::ps} & \stackrel{\text{def}}{=} & v_2\downharpoonleft_{ps} \\ \textit{Stars}\ vs\downharpoonleft_{n::ps} & \stackrel{\text{def}}{=} & vs_{[n]}\downharpoonleft_{ps} \end{array}$$

In the last clause we use Isabelle's notation $vs_{[n]}$ for the *n*th element in a list. The set of positions *inside* a value v, written *Pos* v, is given by

$$\begin{array}{lcl} \textit{Pos}\ (\textit{Empty}) & \stackrel{\text{def}}{=} & \{[]\} \\ \textit{Pos}\ (\textit{Char}\ c) & \stackrel{\text{def}}{=} & \{[]\} \\ \textit{Pos}\ (\textit{Left}\ v) & \stackrel{\text{def}}{=} & \{[]\} \cup \{0 :: ps \mid ps \in \textit{Pos}\ v\} \\ \textit{Pos}\ (\textit{Right}\ v) & \stackrel{\text{def}}{=} & \{[]\} \cup \{1 :: ps \mid ps \in \textit{Pos}\ v\} \\ \textit{Pos}\ (\textit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} & \{[]\} \cup \{0 :: ps \mid ps \in \textit{Pos}\ v_1\} \\ & & \quad \cup \{1 :: ps \mid ps \in \textit{Pos}\ v_2\} \\ \textit{Pos}\ (\textit{Stars}\ vs) & \stackrel{\text{def}}{=} & \{[]\} \cup (\bigcup_{n < len\ vs}\{n :: ps \mid ps \in \textit{Pos}\ vs_{[n]}\}) \end{array}$$

whereby *len* in the last clause stands for the length of a list. Clearly for every position inside a value there exists a subvalue at that position.

To help understanding the ordering of Okui and Suzuki, consider again the earlier value v and compare it with the following w:

$$\begin{array}{rcl} v & \stackrel{\text{def}}{=} & \textit{Stars}\ [\textit{Seq}\ (\textit{Char}\ x)(\textit{Char}\ y),\ \textit{Char}\ z] \\ w & \stackrel{\text{def}}{=} & \textit{Stars}\ [\textit{Char}\ x,\ \textit{Char}\ y,\ \textit{Char}\ z] \end{array}$$

Both values match the string xyz, that means if we flatten these values at their respective root position, we obtain xyz. However, at position $[0]$, v matches xy whereas w matches only the shorter x. So according to the Longest Match Rule, we should prefer v, rather than w as POSIX value for the string xyz (and corresponding regular expression). In order to formalise this idea, Okui and Suzuki introduce a measure for subvalues at position p, called the *norm* of v at position p. We can define this measure in Isabelle/HOL as an integer as follows:

Definition 10 (The Norm of a Value)**.** Given v and p, the norm is defined as

$$\|v\|_p \stackrel{\text{def}}{=} \textit{if}\ p \in \textit{Pos}\ v\ \textit{then}\quad \textit{len}\ |v\downharpoonleft_p|\quad \textit{else}\ -1$$

where we take the length of the flattened value at position p, provided the position is inside v; if it is not inside, then the norm is -1.

The default -1 for outside positions is crucial for the POSIX requirement of preferring a $\textit{Left}$-value over a $\textit{Right}$-value (if they can match the same string—see the Priority Rule from Section 1.2). To see this, consider

$$v \stackrel{\text{def}}{=} \textit{Left}\ (\textit{Char}\ x) \qquad \text{and} \qquad w \stackrel{\text{def}}{=} \textit{Right}\ (\textit{Char}\ x)$$

Both values match x. At position $[0]$ the norm of v is *1* (the subvalue matches x), but the norm of w is *-1* (the position is outside w according to how we defined the 'inside' positions of $\textit{Left}$- and $\textit{Right}$-values). Of course at position $[1]$, the norms $\|v\|_{[1]}$ and $\|w\|_{[1]}$ are reversed, but the point is that subvalues will be analysed according to lexicographically ordered positions. According to this ordering, the position $[0]$ takes precedence over $[1]$ and thus also v will be preferred over w. The lexicographic ordering of positions, written $_ \prec_{lex} _$, can be conveniently formalised by three inference rules

$$\frac{}{[] \prec_{lex} p :: ps} \qquad \frac{p_1 < p_2}{p_1 :: ps_1 \prec_{lex} p_2 :: ps_2} \qquad \frac{ps_1 \prec_{lex} ps_2}{p_1 :: ps_1 \prec_{lex} p :: ps_2}$$

With the norm and lexicographic order in place, we can state the key definition of Okui and Suzuki [41]:

Definition 11. A value v_1 is *smaller at position* p than v_2, written $v_1 \prec_p v_2$, if and only if (i) the norm at position p is greater in v_1 (that is the string $|v_1\downharpoonleft_p|$ is longer than $|v_2\downharpoonleft_p|$) and (ii) all subvalues at positions that are inside v_1 or v_2 and that are lexicographically smaller than p, we have the same norm, namely

$$v_1 \prec_p v_2 \stackrel{\text{def}}{=} \begin{cases} (i) \quad \|v_2\|_p < \|v_1\|_p \qquad \textit{and} \\ (ii) \quad \forall q \in \textit{Pos}\ v_1 \cup \textit{Pos}\ v_2.\ q \prec_{lex} p \rightarrow \|v_1\|_q = \|v_2\|_q \end{cases}$$

The position p in this definition acts as the *first distinct position* of v_1 and v_2, where both values match strings of different length. Since at p the values v_1 and v_2 match different strings, the ordering is irreflexive. Derived from the definition above are the following two auxiliary orderings:

$$v_1 \prec v_2 \stackrel{\text{def}}{=} \exists\, p.\ v_1 \prec_p v_2$$
$$v_1 \preccurlyeq v_2 \stackrel{\text{def}}{=} v_1 \prec v_2 \vee v_1 < v_2$$

While clearly the definition of POSIX values being the minimal values according to this ordering is a "bit more complicated", the point for us is that it is another, independent, specification for POSIX values, and we can actually show that Okui and Suzuki's definition is equivalent to our slightly "less complicated" definition. This is what we shall show next.

Whereas we encountered a number of obstacles for establishing properties like transitivity for the ordering of Sulzmann and Lu (and which we ultimately failed to overcome), it is relatively straightforward to establish this property for the orderings $_ \prec _$ and $_ \preccurlyeq _$ by Okui and Suzuki.

Lemma 7 (Transitivity)**.** *If* $v_1 \prec v_2$ *and* $v_2 \prec v_3$ *then* $v_1 \prec v_3$

Proof. From the assumption we obtain two positions p and q, where the values v_1 and v_2 (respectively v_2 and v_3) are 'distinct'. Since $\prec_{lex}$ is trichotomous, we need to consider three cases, namely $p = q$, $p \prec_{lex} q$ and $q \prec_{lex} p$. Let us look at the first case. Clearly $\|v_2\|_p < \|v_1\|_p$ and $\|v_3\|_p < \|v_2\|_p$ imply that $\|v_3\|_p < \|v_1\|_p$. It remains to show that for a $p' \in \textit{Pos}\ v_1 \cup \textit{Pos}\ v_3$ with $p' \prec_{lex} p$ that $\|v_1\|_{p'} = \|v_3\|_{p'}$ holds. Suppose $p' \in \textit{Pos}\ v_1$, then we can infer from the

first assumption that $\|v_1\|_{p'} = \|v_2\|_{p'}$. But this means that p' must be in $Pos\ v_2$ too (the norm cannot be *-1* given $p' \in Pos\ v_1$). Hence we can use the second assumption and infer $\|v_2\|_{p'} = \|v_3\|_{p'}$, which concludes this case with $v_1 \prec v_3$. The reasoning in the other cases is similar. □

The proof for transitivity of $\preccurlyeq$ is similar and omitted. It is also straightforward to show that $\prec$ and $\preccurlyeq$ are partial orders. Okui and Suzuki [41] furthermore show that they are linear orderings for lexical values of a given regular expression and a given string, but we have not formalised this in Isabelle as it is a bit "hairy" argument. This property is not essential for our result. What we are going to show below is that for a given r and s, the orderings have a unique minimal element in the set $LV\, r\, s$, which is the POSIX value we defined in the previous section by an inductive definition (and which we have already shown to be generated by Sulzmann an Lu's algorithm). We start with two properties that show how the length of a flattened value relates to the $\prec$-ordering.

Proposition 2.

(1) *If* $v_1 \prec v_2$ *then len* $|v_2| \leq$ *len* $|v_1|$.

(2) *If len* $|v_2| < len\ |v_1|$ *then* $v_1 \prec v_2$.

Both properties follow from the definition of Okui and Suzuki's ordering. Note that *(2)* entails that a value, say v_2, whose underlying string is a strict prefix of another flattened value, say v_1, then v_1 must be smaller than v_2. Put in another way, for a given string and a regular expression, a shorter flattened value cannot be a POSIX value. For our proofs it will be useful to have the following properties—in each case the underlying strings of the compared values are the same:

Proposition 3.

(1) *If* $|v_1| = |v_2|$ *then* $\textit{Left}\ v_1 \prec \textit{Right}\ v_2$

(2) *If* $|v_1| = |v_2|$ *then* $\textit{Left}\ v_1 \prec \textit{Left}\ v_2$ *iff* $v_1 \prec v_2$

(3) *If* $|v_1| = |v_2|$ *then* $\textit{Right}\ v_1 \prec \textit{Right}\ v_2$ *iff* $v_1 \prec v_2$

(4) *If* $|v_2| = |w_2|$ *then* $\textit{Seq}\ v\ v_2 \prec \textit{Seq}\ v\ w_2$ *iff* $v_2 \prec w_2$

(5) *If* $|v_1| \mathbin{@} |v_2| = |w_1| \mathbin{@} |w_2|$ *and* $v_1 \prec w_1$ *then*

$$\textit{Seq}\ v_1\ v_2 \prec \textit{Seq}\ w_1\ w_2$$

(6) *If* $|vs_1| = |vs_2|$ *then*

$$\textit{Stars}\ (vs \mathbin{@} vs_1) \prec \textit{Stars}\ (vs \mathbin{@} vs_2)\ \textit{iff}\ \textit{Stars}\ vs_1 \prec \textit{Stars}\ vs_2$$

(7) *If* $|v_1 :: vs_1| = |v_2 :: vs_2|$ *and* $v_1 \prec v_2$ *then*

$$\textit{Stars}\ (v_1 :: vs_1) \prec \textit{Stars}\ (v_2 :: vs_2)$$

One might prefer that statements *(4)* and *(5)* (respectively *(6)* and *(7)*) are combined into a single *iff*-statement (like the ones for *Left* and *Right*). Unfortunately this cannot be done easily: such a single statement would require an additional assumption about the two values *Seq* $v_1 v_2$ and *Seq* $w_1 w_2$ being inhabited by the same regular expression. The complexity of the proofs involved seems to not justify such a 'cleaner' single statement. The statements given above are just the properties that allow us to establish our theorems without any difficulty. The proofs for our version of Proposition 3 are routine.

Next we establish how Okui and Suzuki's orderings relate to our definition of POSIX values. Given a POSIX value v_1 for r and s, then any other lexical value v_2 in $LV\ r\ s$ is greater or equal than v_1, namely:

Theorem 3. *If* $(s,\ r) \rightarrow v_1$ *and* $v_2 \in LV\ r\ s$ *then* $v_1 \preccurlyeq v_2$.

Proof. By induction on our POSIX rules. By Theorem 1 and the definition of *LV*, it is clear that v_1 and v_2 have the same underlying string s. The three base cases are straightforward: for example for $v_1 = \textit{Empty}$, we have that $v_2 \in LV\ \mathbf{1}\ []$

must also be of the form $v_2 = \textit{Empty}$. Therefore we have $v_1 \preccurlyeq v_2$. The inductive cases for r being of the form $r_1 + r_2$ and $r_1 \cdot r_2$ are as follows:

- Case $P + L$ with $(s,\ r_1 + r_2) \rightarrow \textit{Left}\ w_1$: In this case the value v_2 is either of the form *Left* w_2 or *Right* w_2. In the latter case we can immediately conclude with $v_1 \preccurlyeq v_2$ since a *Left*-value with the same underlying string s is always smaller than a *Right*-value by Proposition 3*(1)*. In the former case we have $w_2 \in LV\ r_1\ s$ and can use the induction hypothesis to infer $w_1 \preccurlyeq w_2$. Because w_1 and w_2 have the same underlying string s, we can conclude with *Left* $w_1 \preccurlyeq$ *Left* w_2 using Proposition 3*(2)*.

- Case $P + R$ with $(s,\ r_1 + r_2) \rightarrow \textit{Right}\ w_1$: This case is similar to the previous case, except that we additionally know $s \notin L(r_1)$. This is needed when v_2 is of the form *Left* w_2. Since $|v_2| = |w_2| = s$ and $\vdash w_2 : r_1$, we can derive a contradiction for $s \notin L(r_1)$ using Proposition 1. So also in this case $v_1 \preccurlyeq v_2$.

- Case PS with $(s_1 @ s_2, r_1 \cdot r_2) \rightarrow \textit{Seq}\ w_1\ w_2$: We can assume $v_2 = \textit{Seq}$ $u_1\ u_2$ with $u_1 : r_1$ and $u_2 : r_2$. We have $s_1 @ s_2 = |u_1| @ |u_2|$. By the side-condition of the *PS*-rule we know that either $s_1 = |u_1|$ or that $|u_1|$ is a strict prefix of s_1. In the latter case we can infer $w_1 \prec u_1$ by Proposition 2*(2)* and from this $v_1 \preccurlyeq v_2$ by Proposition 3*(5)* (as noted above v_1 and v_2 must have the same underlying string). In the former case we know $u_1 \in LV\ r_1\ s_1$ and $u_2 \in LV\ r_2\ s_2$. With this we can use the induction hypotheses to infer $w_1 \preccurlyeq u_1$ and $w_2 \preccurlyeq u_2$. By Proposition 3*(4,5)* we can again infer $v_1 \preccurlyeq v_2$.

The case for P^* is similar to the PS-case and omitted. □

This theorem shows that our POSIX value for a regular expression r and a string s is in fact a minimal element of the values in $LV\ r\ s$. By Proposition 2*(2)*

we also know that any value in $LV\ r\ s'$, with s' being a strict prefix, cannot be smaller than v_1.

The next theorem shows the opposite—namely any minimal element in $LV\ r\ s$ must be a POSIX value. This can be established by induction on r, but the proof can be drastically simplified by using the fact from the previous section about the existence of a POSIX value whenever a string $s \in L(r)$.

Theorem 4. *If* $v_1 \in LV\ r\ s$ *and* $(\forall\, v_2 \in LV\ r\ s.\ v_2 \not\prec v_1)$ *then* $(s,\ r) \to v_1$.

Proof. If $v_1 \in LV\ r\ s$ then $s \in L(r)$ by Proposition 1. Hence by Theorem 2*(2)* there exists a POSIX value v_p with $(s,\ r) \to v_p$ and by Lemma 4 we also have $v_p \in LV\ r\ s$. By Theorem 3 we therefore have $v_p \preccurlyeq v_1$. If $v_p = v_1$ then we are done. Otherwise we have $v_p \prec v_1$, which however contradicts the second assumption about v_1 being the smallest element in $LV\ r\ s$. So we are done in this case too. □

From this we can also show that if $LV\ r\ s$ is non-empty (or equivalently $s \in L(r)$) then it has a unique minimal element and it is in fact the one returned by the *lexer*.

Corollary 2. *If* $LV\ r\ s \neq \varnothing$ *then*

$$\exists! v_{min} \in LV\, r\, s.\ lexer = Some\,(v_{min}) \wedge (\forall v \in LV\, r\, s.\ \ v_{min} \preccurlyeq v)$$

To sum up, we have shown that the (unique) minimal elements of the ordering by Okui and Suzuki are exactly the POSIX values we defined inductively in Section 2.1. This provides an independent confirmation that our ternary relation formalises indeed the informal POSIX rules.

Since the idea behind the ordering by Okui and Suzuki is somewhat similar to the idea behind the ordering of Sulzmann and Lu, our hope was we would finally be in a position to make some progress with formalising their correctness

$$\frac{v_1 \succcurlyeq_{gr} v_1'}{\mathit{Seq}(v_1, v_2) \succcurlyeq_{gr} \mathit{Seq}(v_1', v_2')}GS_1 \qquad \frac{v_2 \succcurlyeq_{gr} v_2'}{\mathit{Seq}(v_1, v_2) \succcurlyeq_{gr} \mathit{Seq}(v_1, v_2')}GS_2$$

$$\frac{v_1 \succcurlyeq_{gr} v_2}{\mathit{Left}\ v_1 \succcurlyeq_{gr} \mathit{Left}\ v_2}GLL \qquad \frac{v_1 \succcurlyeq_{gr} v_2}{\mathit{Right}\ v_1 \succcurlyeq_{gr} \mathit{Right}\ v_2}GRR$$

$$\frac{}{\mathit{Left}\ v_2 \succcurlyeq_{gr} \mathit{Right}\ v_1}GLR$$

$$\frac{v_1 \succcurlyeq_{gr} v_2}{\mathit{Stars}\ (v_1 :: vs_1) \succcurlyeq_{gr} \mathit{Stars}\ (v_2 :: vs_2)}G{*}_1$$

$$\frac{vs_1 \succcurlyeq_{gr} vs_2}{\mathit{Stars}\ (v :: vs_1) \succcurlyeq_{gr} \mathit{Stars}\ (v :: vs_2)}G{*}_2$$

$$\frac{}{\mathit{Stars}\ (v :: vs) \succcurlyeq_{gr} \mathit{Stars}\ []}G{*}_3$$

$$\frac{}{\mathit{Char}\ c \succcurlyeq_{gr} \mathit{Char}\ c}GC \qquad \frac{}{\mathit{Empty} \succcurlyeq_{gr} \mathit{Empty}}GE$$

Figure 2.2: The reflexive version of the ordering by Frisch and Cardelli for GREEDY matching.

proof. Alas this turned out not to be true. Therefore let us next go back to the GREEDY ordering introduced by Frisch and Cardelli from where the Sulzmann and Lu took the proof idea. And then analyse in detail where we think Sulzmann and Lu's proof breaks down.

2.3 GREEDY Ordering by Frisch and Cardelli

Frisch and Cardelli [23] introduced an ordering, written $\succcurlyeq_{gr}$, for values and they show that their GREEDY matching algorithm always produces a maximal element according to this ordering (from all possible solutions). Their ordering $\succcurlyeq_{gr}$ is defined by the rules shown in Figure 2.2. The only difference between our version of their rules and their original rules is that we made the relation reflexive by including rules GC and GE. But this is a harmless addition.

That these rules realise a GREEDY ordering can be seen in the GLR rule

where a Left-value is always bigger than (or preferred over) a Right-value. What is interesting for our purposes here is that the properties reflexivity, totality and transitivity for this GREEDY ordering can be proved relatively easily by inductions. This is illustrated next:

Lemma 8 (Reflexivity)**.** If $\vdash v : r$ then $v \succcurlyeq_{gr} v$.

Proof. This is by a straightforward induction on the definition of $\vdash v : r$. □

Lemma 9 (Totality)**.** If $\vdash v_1 : r$ and $\vdash v_2 : r$ then $v_1 \succcurlyeq_{gr} v_2$ or $v_2 \succcurlyeq_{gr} v_1$.

Proof. This is again by a straightforward induction on the definition of $\vdash v_1 : r$ and a case-analysis of $\vdash v_2 : r$. □

We can also show transitivity by induction on r.

Lemma 10 (Transitivity)**.** Suppose $\vdash v_1 : r$, $\vdash v_2 : r$ and $\vdash v_3 : r$. If $v_1 \succcurlyeq_{gr} v_2$ and $v_2 \succcurlyeq_{gr} v_3$ then $v_1 \succcurlyeq_{gr} v_3$.

Proof. By induction on r analysing all cases of $\vdash v_1 : r$ and so on. The only interesting case is for sequences, where we can assume $v_1 = \mathit{Seq}(v_{1l}, v_{1r})$, $v_2 = \mathit{Seq}(v_{2l}, v_{2r})$, and $v_3 = \mathit{Seq}(v_{3l}, v_{3r})$. We need to show that

$$\mathit{Seq}(v_{1l}, v_{1r}) \succcurlyeq_{gr} \mathit{Seq}(v_{3l}, v_{3r})$$

holds under the assumptions that $\mathit{Seq}(v_{1l}, v_{1r}) \succcurlyeq_{gr} \mathit{Seq}(v_{2l}, v_{2r})$ holds and that $\mathit{Seq}(v_{2l}, v_{2r}) \succcurlyeq_{gr} \mathit{Seq}(v_{3l}, v_{3r})$ holds. There are two rules which could have derived each assumption. For example $v_{1l} \succcurlyeq_{gr} v_{2l}$ and $v_{2l} \succcurlyeq_{gr} v_{3l}$. In this case we can apply the induction hypothesis and derive $v_{1l} \succcurlyeq_{gr} v_{3l}$ from this we obtain $\mathit{Seq}(v_{1l}, v_{1r}) \succcurlyeq_{gr} \mathit{Seq}(v_{3l}, v_{3r})$. The other three cases are similar (where in one case we need to appeal to the reflexivity property). □

It should not come as a surprise that if we make changes to the ordering rules (unless they are really harmless, like our addition of rules GC and GE), the proof ideas behind these proofs might not necessarily transfer to the modified rules. That is what we shall show in the next section about the POSIX ordering rules introduced by Sulzmann and Lu.

2.4 POSIX Ordering by Sulzmann and Lu

As mentioned before, the rules by Sulzmann and Lu [53] are a variant of the GREEDY rules by Frisch and Cardelli. One difference is that Sulzmann and Lu's ordering, written $\succcurlyeq^r_{PX}$, also includes a regular expression. The rules are shown in Figure 2.3. The only difference between the original rules by Sulzmann and Lu, and the ones shown is the inclusion of the rules C and E which make the ordering reflexive (that is reflexivity is directly built into the inductive definition of the ordering, rather than an auxiliary definition as in Sulzmann and Lu). We also slightly adapted their notation to fit our conventions.

The interesting rules are $A1$ and $A2$. For this remember that the GREEDY ordering always prefers a Left-value over a Right-value. This is different in the POSIX rules: there a Right-value is preferred provided it can match a longer string (the $A1$ rule); a Left-value is only preferred when it can match a longer or equal string than the Right-value (the $A2$ rule). Perhaps surprisingly, but perhaps not, this "small" change in the Sulzmann and Lu's rules has drastic consequences for the proofs.

To start with, transitivity does not hold anymore in the "normal" formulation, that is:

Property 1. Suppose $\vdash v_1 : r$, $\vdash v_2 : r$ and $\vdash v_3 : r$. If $v_1 \succcurlyeq^r_{PX} v_2$ and $v_2 \succcurlyeq^r_{PX} v_3$ then $v_1 \succcurlyeq^r_{PX} v_3$.

$$\frac{v_1 \succcurlyeq^{r_1}_{PX} v_1'}{\mathit{Seq}(v_1, v_2) \succcurlyeq^{r_1 \cdot r_2}_{PX} \mathit{Seq}(v_1', v_2')} C2 \qquad \frac{v_2 \succcurlyeq^{r_2}_{PX} v_2'}{\mathit{Seq}(v_1, v_2) \succcurlyeq^{r_1 \cdot r_2}_{PX} \mathit{Seq}(v_1, v_2')} C1$$

$$\frac{\mathit{len}\,|v_2| > \mathit{len}\,|v_1|}{\mathit{Right}\ v_2 \succcurlyeq^{r_1 + r_2}_{PX} \mathit{Left}\ v_1} A1 \qquad \frac{\mathit{len}\,|v_1| \geq \mathit{len}\,|v_2|}{\mathit{Left}\ v_1 \succcurlyeq^{r_1 + r_2}_{PX} \mathit{Right}\ v_2} A2$$

$$\frac{v_2 \succcurlyeq^{r_2}_{PX} v_2'}{\mathit{Right}\ v_2 \succcurlyeq^{r_1 + r_2}_{PX} \mathit{Right}\ v_2'} A3 \qquad \frac{v_1 \succcurlyeq^{r_1}_{PX} v_1'}{\mathit{Left}\ v_1 \succcurlyeq^{r_1 + r_2}_{PX} \mathit{Left}\ v_1'} A4$$

$$\frac{|v :: vs| = []}{\mathit{Stars}\ [] \succcurlyeq^{r^*}_{PX} \mathit{Stars}\ (v :: vs)} K1 \qquad \frac{|v :: vs| \neq []}{\mathit{Stars}\ (v :: vs) \succcurlyeq^{r^*}_{PX} \mathit{Stars}\ []} K2$$

$$\frac{v_1 \succcurlyeq^{r}_{PX} v_2}{\mathit{Stars}\ (v_1 :: vs_1) \succcurlyeq^{r^*}_{PX} \mathit{Stars}\ (v_2 :: vs_2)} K3$$

$$\frac{\mathit{Stars}\ vs_1 \succcurlyeq^{r^*}_{PX} \mathit{Stars}\ vs_2}{\mathit{Stars}\ (v :: vs_1) \succcurlyeq^{r^*}_{PX} \mathit{Stars}\ (v :: vs_2)} K4$$

$$\frac{}{\mathit{Char}\ c \succcurlyeq^{c}_{PX} \mathit{Char}\ c} C \qquad \frac{}{\mathit{Empty} \succcurlyeq^{\mathbf{1}}_{PX} \mathit{Empty}} E$$

Figure 2.3: The reflexive version of the ordering by Sulzmann and Lu for POSIX matching.

If formulated like this, then there are various counter examples: Suppose r is $a + ((a + a) \cdot (a + \mathbf{1}))$ then the v_1, v_2 and v_3 below are values of r:

$$\begin{aligned} v_1 &= \mathit{Left}(\mathit{Char}\ a) \\ v_2 &= \mathit{Right}(\mathit{Seq}(\mathit{Left}(\mathit{Char}\ a), \mathit{Right}(\mathit{Empty}))) \\ v_3 &= \mathit{Right}(\mathit{Seq}(\mathit{Right}(\mathit{Char}\ a), \mathit{Left}(\mathit{Char}\ a))) \end{aligned}$$

Moreover $v_1 \succcurlyeq^{r}_{PX} v_2$ and $v_2 \succcurlyeq^{r}_{PX} v_3$, but *not* $v_1 \succcurlyeq^{r}_{PX} v_3$! The reason is that although v_3 is a Right-value, it can match a longer string, namely $|v_3| = aa$, while $|v_1|$ (and $|v_2|$) matches only a. So transitivity in this formulation does *not* hold—in this example actually $v_3 \succcurlyeq^{r}_{PX} v_1$!

Sulzmann and Lu "fix" this problem by weakening the transitivity property. They require in addition that the underlying strings are of the same length. This

excludes the counter example above and any counter-example we could find with an implementation. Thus the transitivity lemma in [53] is:

Property 2. Suppose $\vdash v_1 : r$, $\vdash v_2 : r$ and $\vdash v_3 : r$, and also $|v_1| = |v_2| = |v_3|$. If $v_1 \succcurlyeq^{r}_{PX} v_2$ and $v_2 \succcurlyeq^{r}_{PX} v_3$ then $v_1 \succcurlyeq^{r}_{PX} v_3$.

While we agree with Sulzmann and Lu that this property probably holds, proving it seems not so straightforward. Sulzmann and Lu do not give an explicit proof of the transitivity property, but give a closely related property about the existence of maximal elements. They state that this can be verified by an induction on r. We disagree with this as we shall show next in case of transitivity.

The case where the reasoning breaks down is the sequence case, say $r_1 \cdot r_2$. The induction hypotheses in this case are

IH r_1:

$$\begin{array}{ll} \forall v_1, v_2, v_3. & \vdash v_1 : r_1 \wedge \\ & \vdash v_2 : r_1 \wedge \\ & \vdash v_3 : r_1 \wedge \\ & |v_1| = |v_2| = |v_3| \wedge \\ & v_1 \succcurlyeq^{r_1}_{PX} v_2 \;\wedge\; v_2 \succcurlyeq^{r_1}_{PX} v_3 \\ & \quad \Rightarrow v_1 \succcurlyeq^{r_1}_{PX} v_3 \end{array}$$

IH r_2:

$$\begin{array}{ll} \forall v_1, v_2, v_3. & \vdash v_1 : r_2 \wedge \\ & \vdash v_2 : r_2 \wedge \\ & \vdash v_3 : r_2 \wedge \\ & |v_1| = |v_2| = |v_3| \wedge \\ & v_1 \succcurlyeq^{r_2}_{PX} v_2 \;\wedge\; v_2 \succcurlyeq^{r_2}_{PX} v_3 \\ & \quad \Rightarrow v_1 \succcurlyeq^{r_2}_{PX} v_3 \end{array}$$

We can assume that

$$Seq(v_{1l}, v_{1r}) \succcurlyeq^{r_1 \cdot r_2}_{PX} Seq(v_{2l}, v_{2r}) \qquad \text{and} \qquad Seq(v_{2l}, v_{2r}) \succcurlyeq^{r_1 \cdot r_2}_{PX} Seq(v_{3l}, v_{3r}) \tag{2.1}$$

hold, and furthermore that the values have equal length, namely:

$$|Seq(v_{1l}, v_{1r})| = |Seq(v_{2l}, v_{2r})| \quad \text{and} \quad |Seq(v_{2l}, v_{2r})| = |Seq(v_{3l}, v_{3r})| \tag{2.2}$$

We need to show that

$$Seq(v_{1l}, v_{1r}) \succcurlyeq^{r_1 \cdot r_2}_{PX} Seq(v_{3l}, v_{3r})$$

holds. We can proceed by analysing how the assumptions in (2.1) have arisen. There are four cases. Let us assume we are in the case where we know

$$v_{1l} \succcurlyeq^{r_1}_{PX} v_{2l} \qquad \text{and} \qquad v_{2l} \succcurlyeq^{r_1}_{PX} v_{3l}$$

and also know the corresponding typing judgements. This is exactly a case where we would like to apply the induction hypothesis IH r_1. But we cannot! We still need to show that $|v_{1l}| = |v_{2l}|$ and $|v_{2l}| = |v_{3l}|$. We know from (2.2) that the lengths of the sequence values are equal, but from this we cannot infer anything about the lengths of the component values. Indeed in general they will be unequal, that is

$$|v_{1l}| \neq |v_{2l}| \qquad \text{and} \qquad |v_{1r}| \neq |v_{2r}|$$

but still (2.2) will hold. Now we are stuck, since the IH does not apply. This problem where the induction hypothesis does not apply arises in several places in the proof of Sulzmann and Lu, not just for proving transitivity.

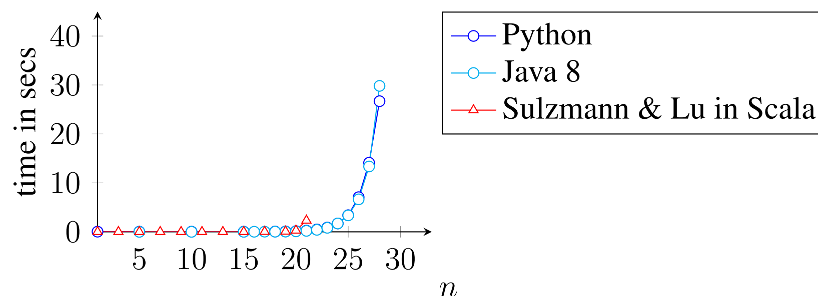
The immediate effect is that the existence of a unique maximal value cannot be inferred. We know totality of $\succcurlyeq^{r}_{PX}$ and know that for every regular expression there are only a finite number of (proper) values. But without transitivity it seems hard to establish that given a regular expression and given a string, there exists always a unique maximal value. . . which the algorithm is supposed to calculate. Without this basic property, the whole correctness proofs already collapses. To sum up, the weakening of the properties by requiring that values need to have equal length seems to make the properties to hold, but destroys all inductive properties in the sequence case. The result is we were not able to formalise any

substantial part of Sulzmann and Lu's "pencil-and-paper" proof.

Chapter 3

Optimisations, Extensions and Future Work

Having been able to prove the correctness of Sulzmann & Lu's algorithm according to Okui & Suzuki's specification and also our own (equivalent) specification of POSIX values, we should now have look at how the algorithm performs in terms of speed. Transliterating (manually) the Isabelle code of the algorithm into Scala, for example, is pleasantly straightforward—this is a major attraction of the algorithm by Brzozowski, and Sulzmann and Lu. Alas, trying out the code on our standard example from the Introduction involving $(a^*)^* \cdot b$ and strings of the form $a \ldots a$ leads to sobering news:



In this rough comparison, the algorithm actually performs worse than the regular expression matchers in Python and Java, which we heavily criticised in the

Introduction for their abysmal runtime behaviour. In fact we can only take measurements for strings up to the length of 21 a's, because with longer strings one consistently obtains “out of memory” exceptions. The problem is that derivatives as calculated by Definition 5 can grow very big and as a result slow down the matching process: every time we call $nullable$ or the derivative function we essentially need to traverse the corresponding regular expression trees and value trees. Therefore, large trees (regular expressions) lead to slow matching. We shall look next at how this problem can be addressed.

3.1 Simplification of Regular Expressions

Derivatives as calculated by Brzozowski's method are usually more complex regular expressions than the initial one; the result is that the derivative-based matching and lexing algorithms are often abysmally slow (see graph above). However, as Sulzmann and Lu wrote, various optimisations are possible, such as the simplifications of $\mathbf{0} + r, r + \mathbf{0}, \mathbf{1} \cdot r$ and $r \cdot \mathbf{1}$ to r. These simplifications can speed up the algorithms considerably [53]. One of the advantages of having a simple specification and correctness proof is that the latter can be extended to also establish the correctness of such simplification steps.

While the simplification of regular expressions according to rules such as

$$\begin{array}{lcl}
\mathbf{0} + r & \Rightarrow & r \\
r + \mathbf{0} & \Rightarrow & r \\
\mathbf{1} \cdot r & \Rightarrow & r \\
r \cdot \mathbf{1} & \Rightarrow & r \\
\mathbf{0} \cdot r & \Rightarrow & \mathbf{0} \\
r \cdot \mathbf{0} & \Rightarrow & \mathbf{0}
\end{array} \tag{3.1}$$

is well-understood, there is an obstacle with the POSIX value calculation algorithm by Sulzmann and Lu: if we build a derivative regular expression and then simplify it, we will calculate a POSIX value for the simplified derivative regular expression, *not* for the original (unsimplified) derivative regular expression. This produces incorrect results. Sulzmann and Lu overcome this obstacle in an early version of [53][1] by not just calculating a simplified regular expression, but also calculating a *rectification function* that "repairs" the incorrect value.

The idea behind the rectification functions is as follows: if we have a regular expression of the form, say, $\mathbf{0} + r$, then we simplify it to just r and calculate the POSIX value for how r matched the corresponding string. Suppose this gives the value v. Then in order to obtain a POSIX value for $\mathbf{0} + r$ we have to rectify v to be $Right(v)$. This would be the same, for example, for regular expressions of the form $r + \mathbf{0}$. In this case we would have to rectify the value from v to $Left(v)$. Similarly for $r \cdot \mathbf{1}$ where we have to rectify a v to $Seq\, v\, Empty$. The only difficulty is that such simplifications can occur deep inside the regular expressions and we need to compose, or "stage", the rectification functions in the right way. Otherwise we break the correctness of the algorithm.

The rectification functions can be (slightly clumsily) implemented in Isabelle using the auxiliary functions shown in Figure 3.1. The functions $simp_{Alt}$ and $simp_{Seq}$ encode the simplification rules shown in (3.1) on page 68 and compose the rectification functions (recall simplifications can occur deep inside a regular expression). The main simplification function *simp* is then defined as

$$\begin{aligned}
simp\ (r_1 + r_2) &\stackrel{\text{def}}{=} simp_{Alt}\ (simp\ r_1)\ (simp\ r_2) \\
simp\ (r_1 \cdot r_2) &\stackrel{\text{def}}{=} simp_{Seq}\ (simp\ r_1)\ (simp\ r_2) \\
simp\ r &\stackrel{\text{def}}{=} (r, id)
\end{aligned}$$

where id stands for the identity function. As can be seen, the function *simp* re-

[1] https://sites.google.com/site/luzhuomi/file/icfp13.pdf

$$
\begin{array}{lcl}
F_{\textit{Right}}\ f\ v & \stackrel{\text{def}}{=} & \textit{Right}\ (f\ v)\\
F_{\textit{Left}}\ f\ v & \stackrel{\text{def}}{=} & \textit{Left}\ (f\ v)\\
F_{\textit{Alt}}\ f_1\ f_2\ (\textit{Right}\ v) & \stackrel{\text{def}}{=} & \textit{Right}\ (f_2\ v)\\
F_{\textit{Alt}}\ f_1\ f_2\ (\textit{Left}\ v) & \stackrel{\text{def}}{=} & \textit{Left}\ (f_1\ v)\\
F_{\textit{Seq1}}\ f_1\ f_2\ v & \stackrel{\text{def}}{=} & \textit{Seq}\ (f_1\ Empty)(f_2\ v)\\
F_{\textit{Seq2}}\ f_1\ f_2\ v & \stackrel{\text{def}}{=} & \textit{Seq}\ (f_1\ v)(f_2\ Empty)\\
F_{\textit{Seq}}\ f_1\ f_2\ (\textit{Seq}\ v_1\ v_2) & \stackrel{\text{def}}{=} & \textit{Seq}\ (f_1\ v_1)(f_2\ v_2)
\end{array}
$$

$$
\begin{array}{lcl}
\textit{simp}_{\textit{Alt}}(\mathbf{0},_)(r_2,f_2) & \stackrel{\text{def}}{=} & (r_2, F_{\textit{Right}}\ f_2)\\
\textit{simp}_{\textit{Alt}}(r_1,f_1)(\mathbf{0},_) & \stackrel{\text{def}}{=} & (r_1, F_{\textit{Left}}\ f_1)\\
\textit{simp}_{\textit{Alt}}(r_1,f_1)(r_2,f_2) & \stackrel{\text{def}}{=} & (r_1+r_2, F_{\textit{Alt}}\ f_1\ f_2)\\
\textit{simp}_{\textit{Seq}}(\mathbf{1},f_1)(r_2,f_2) & \stackrel{\text{def}}{=} & (r_2, F_{\textit{Seq1}}\ f_1\ f_2)\\
\textit{simp}_{\textit{Seq}}(r_1,f_1)(\mathbf{1},f_2) & \stackrel{\text{def}}{=} & (r_1, F_{\textit{Seq2}}\ f_1\ f_2)\\
\textit{simp}_{\textit{Seq}}(\mathbf{0},f_1)(r_2,f_2) & \stackrel{\text{def}}{=} & (\mathbf{0}, \textit{undefined})\\
\textit{simp}_{\textit{Seq}}(r_1,f_1)(\mathbf{0},f_2) & \stackrel{\text{def}}{=} & (\mathbf{0}, \textit{undefined})\\
\textit{simp}_{\textit{Seq}}(r_1,f_1)(r_2,f_2) & \stackrel{\text{def}}{=} & (r_1\cdot r_2, F_{\textit{Seq}}\ f_1\ f_2)
\end{array}
$$

Figure 3.1: Auxiliary functions for simplifying regular expressions and rectifying values. In the cases where the simplification yields **0**, we can specify the rectification function as *undefined* as it will never been called during matching.

turns a simplified regular expression and also a corresponding rectification function. Note that we do not simplify under stars: this seems to slow down the algorithm, rather than speed it up. The optimised lexer can then be given by the clauses:

$$
\begin{array}{lcl}
\textit{slexer}\ r\ [] & \stackrel{\text{def}}{=} & \textit{if nullable}\ r\ \textit{then Some}\ (\textit{mkeps}\ r)\ \textit{else None}\\
\textit{slexer}\ r\ (c::s) & \stackrel{\text{def}}{=} & \textit{let}\ (r_{simp}, f_{rect}) = \textit{simp}\ (r\backslash c)\ \textit{in}\\
& & \textit{case slexer}\ r_{simp}\ s\ \textit{of}\\
& & \quad \textit{None} \Rightarrow \textit{None}\\
& & \mid \textit{Some}\ v \Rightarrow \textit{Some}\ (\textit{inj}\ r\ c\ (f_{rect}\ v))
\end{array}
$$

The first clause is unchanged from *lexer*. In the second clause we first calculate the derivative $r\backslash c$ and then simplify the result. This gives us a simplified derivative r_{simp} and a rectification function f_{rect}. The lexer is then recursively

called with the simplified derivative and the shorter string where the character c is chopped off. The point is that when we receive back a value, say v, for the simplified derivative, we need to rectify v (that is construct $f_{rect}\, v$) before injecting the character c back into the rectified value.

In order to establish the correctness of *slexer*, we need to show that simplification preserves the language and simplification preserves our POSIX relation, provided the value is rectified. To see what is going on in the next lemma, recall that *simp* generates a (regular expression, rectification function) pair. In the first property we show that every regular expression is equivalent to its simplified version (that is matches the same language). In the second we show that if we obtain a value for a simplified regular expression and it is a POSIX value, then if we rectify the value, it will be a POSIX value for the original (unsimplified) regular expression.

Lemma 11.

(1) $L\,(\mathit{fst}\,(\mathit{simp}\,r)) = L(r)$

(2) *If* $(s, \mathit{fst}\,(\mathit{simp}\,r)) \rightarrow v$ *then* $(s, r) \rightarrow \mathit{snd}\,(\mathit{simp}\,r)\,v$.

Proof. Both are by induction on r. There is no interesting case for the first statement. For the second statement, of interest are the $r = r_1 + r_2$ and $r = r_1 \cdot r_2$ cases. In each case we have to analyse four subcases whether $\mathit{fst}\ (\mathit{simp}\ r_1)$ and $\mathit{fst}\ (\mathit{simp}\ r_1)$ equals $\mathbf{0}$ (respectively $\mathbf{1}$). For example for $r = r_1 + r_2$, consider the subcase $\mathit{fst}\ (\mathit{simp}\ r_1) = \mathbf{0}$ and $\mathit{fst}\ (\mathit{simp}\ r_2) \neq \mathbf{0}$. By assumption we know $(s, \mathit{fst}$ $(\mathit{simp}\ (r_1 + r_2))) \rightarrow v$. From this we can infer $(s, \mathit{fst}\ (\mathit{simp}\ r_2)) \rightarrow v$ and by IH also $(*)(s, r_2) \rightarrow \mathit{snd}\ (\mathit{simp}\ r_2)\ v$. Given $\mathit{fst}\ (\mathit{simp}\ r_1) = \mathbf{0}$, we know $L\ (\mathit{fst}\ (\mathit{simp}$ $r_1)) = \varnothing$. By the first statement $L(r_1)$ is the empty set, meaning (**) $s \notin L(r_1)$. Taking (*) and (**) together gives by the *P*+*R*-rule $(s,\ r_1 + r_2) \rightarrow \mathit{Right}\ (\mathit{snd}$ $(\mathit{simp}\ r_2)\ v)$. In turn this gives $(s, r_1 + r_2) \rightarrow \mathit{snd}\ (\mathit{simp}\ (r_1 + r_2))\ v$ as we need to show. The other cases are similar. □

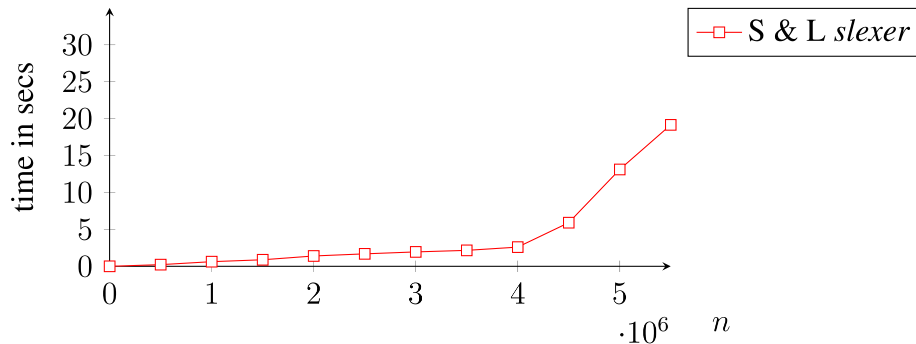
We can now prove relatively straightforwardly that the optimised lexer produces the expected result:

Theorem 5. *slexer* r s $=$ *lexer* r s

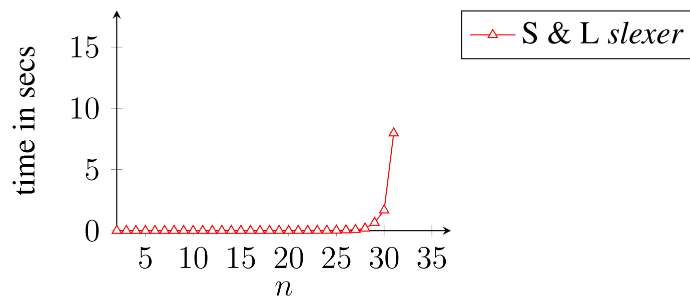
Proof. By induction on s generalising over r. The [] case is trivial. For the cons-case suppose the string is of the form $c :: s$. By induction hypothesis we know *slexer* r s $=$ *lexer* r s holds for all r (in particular for r being the derivative $r\backslash c$). Let r_s be the simplified derivative regular expression, that is *fst* $($*simp* $(r\backslash c))$, and f_r be the rectification function, that is *snd* $($*simp* $(r\backslash c))$. We distinguish the cases whether (*) $s \in L(r\backslash c)$ or not. In the first case we have by Theorem 2(2) a value v so that *lexer* $(r\backslash c)$ s $=$ *Some* v and $(s, r\backslash c) \to v$ hold. By Lemma 11(1) we can also infer from (*) that $s \in L(r_s)$ holds. Hence we know by Theorem 2(2) that there exists a v' with *lexer* r_s s $=$ *Some* v' and $(s,\ r_s) \to v'$. From the latter we know by Lemma 11(2) that $(s,\ r\backslash c) \to f_r\ v'$ holds. By the uniqueness of the POSIX relation (Theorem 1) we can infer that v is equal to $f_r\ v'$—that is the rectification function applied to v' produces the original v. Now the case follows by the definitions of *lexer* and *slexer*.

In the second case where $s \notin L\ (r\backslash c)$ we have that *lexer* $(r\backslash c)$ s $=$ *None* by Theorem 2(1). We also know by Lemma 11(1) that $s \notin L\ (r_s)$. Hence *lexer* r_s s $=$ *None* by Theorem 2(1) and by IH then also *slexer* r_s s $=$ *None*. With this we can conclude in this case too. □

Having this correctness result under our belt, we can perform again some rough calculations with *slexer* in Scala. This time we obtain more promising results. In the example with $(a^*)^* \cdot b$ we can now process strings up to 5.5 Million(!) a's in just under 20 seconds (in Java and Python we were only able to process strings up to 30 a's).



The reason for this good performance is that in this example the simplification process keeps the derivative regular expression to a nearly constant size (there is usually an initial small growth until a "fix point" is reached after which the size of the derivatives is either constant or decreases). Trying out *slexer* on more examples and even lexing some small toy programming languages shows that it gives decent processing times in many instances. Unfortunately there are examples where the picture is not as rosy as one might wish. For instance in the small example $(a + aa)^*$ the derivative after 31 a's contains already more than 20 Million nodes (despite the simplification in *slexer*) and this clearly affects the running time as shown in the graph below:



The reason for this disappointing result is that our simplification is "local" in the sense that it descends regular expressions towards the inside, but only simpli-

fies locally on the way up. It does not perform "global" rewrites, because then defining appropriate rectification functions and staging them correctly becomes significantly harder. Unfortunately the derivative function and the local simplification method produce in the example $(a+aa)^*$ intermediate regular expressions of the form

$$(r + r') + r$$

To deal with such instances the simplification function would have to find out whether there is a r and then later on (to the right) there is another regular expression of exactly the same shape. Since it comes later, or more precisely further to the right-hand side, we know that the second occurrence of r cannot contribute to the POSIX value, because of the Priority Rule in POSIX. So it could be safely deleted. But even if we include a simplification rule like $r + r \Rightarrow r$ in *slexer* this problem cannot be solved, because there can be "anything" in between the first and second occurrence of r, as indicated above.

While we have managed to make some initial progress towards a more enhanced simplification of regular expressions in Scala(!) code, we have *not* managed to obtain anything simple enough in order to start proving the correctness for such a more enhanced simplification function. Also we are not sure (that is we have no proof) whether this Scala code satisfies the property that for every regular expression and for every string there is a kind of "fixpoint" after which derivatives do not grow bigger. This is a property one ultimately wants to achieve in order to have an efficient derivative-based lexing algorithm. After some intellectual "zig-zagging", we found that going back to Sulzmann and Lu's paper [53] helped us with addressing this "speed" problem by using bitcoded values and annotated regular expressions.

3.2 Bitcoded Values and Annotated Regular Expressions

In the second part of their paper [53],[2] Sulzmann and Lu introduce a bitcoded version of their lexing algorithm. They make some claims about the correctness and speed of this version, but do not provide any supporting proof arguments, not even "pencil-and-paper" arguments. They wrote about their bitcoded "incremental parsing method" (that is the algorithm to be studied in this section):

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

We shall make partial progress in this section by supplying one important part of the missing proofs. We shall show that the incremental construction of values *without* simplification is correct. There is already recent work by Ribeiro and Du Bois [48] in Agda on this topic. They present some formalised proofs about bitcoded regular expression matching and derivatives, but we found they do *not* address the more important problem of whether Sulzmann and Lu's bitcoded algorithm produces correct results.

The values generated by Sulzmann and Lu's original algorithm can be seen as trees that need to be represented appropriately in memory. If they are represented as trees (or inductive datatypes) then clearly this results in significant memory requirements. So it seems self-evident that a more compact representation, for

[2] This refers to the "final" version of the paper that appeared in the FLOPS'14 conference proceedings. There is also a more recent and extended version available from the first author's webpage where some of our concerns about the proof are addressed.

example, as bitcoded sequences, is preferable. While the bitcoding of values introduced by Sulzmann and Lu looks at first glance as just an improvement in terms of memory, rather than speed, this first appearance is deceiving. In fact, the idea of representing values as bit-sequences and annotating them in regular expressions is a very clever design that makes proving the correctness of a more powerful simplification method feasible. That is probably also the reason why Sulzmann and Lu switched from the simplification/rectification technique, which we have discussed and proved correct in the previous section, to the technique of using bitcoded values/annotated regular expressions in their published version of [53].

We shall provide here a proof for the claim by Sulzmann and Lu that the *un*simplified version of their bitcoded algorithm produces correct results. This is a key stepping stone for establishing the correctness of an algorithm involving more "aggressive" simplification rules. While this is only a partial result, it is still significant progress, because the bitcoded algorithm builds values incrementally and from the "wrong" end, in comparison with the "standard" way how *lexer* constructs values. To see the difference, recall that Sulzmann and Lu's *lexer* consists of *two* phases, see Figure 1.1 on Page 42—a derivative building phase and a subsequent value building phase. The bitcoded algorithm, in contrast, only consist of a single phase. Each derivative step will already generate, incrementally, some parts of the final value (represented as bit-sequence).

For giving our proof, let us start with an auxiliary function *flex* that allows us to recast the rules of $lexer$ (with its two phases) in terms of a single phase.

Definition 12.

$$\begin{array}{lcl} \textit{flex}\; r\; f\; [] & \stackrel{\text{def}}{=} & f \\ \textit{flex}\; r\; f\; (c :: s) & \stackrel{\text{def}}{=} & \textit{flex}\; (r\backslash c)\; (\lambda v.\, f\; (\textit{inj}\; r\; c\; v))\; s \end{array}$$

The point of this function is to do lexing in a "forward" manner where we stack

up injection functions while building derivatives. When reaching the end of the string, we just need to apply the stacked injection functions to the value generated by $mkeps$. Using this function we can recast the definition of *lexer* as follows:

Lemma 12.

$$\begin{array}{lcl} \textit{lexer}\, r\, s & = & \textit{if}\ nullable(r\backslash s) \\ & & \textit{then Some}(\textit{flex}\, r\, id\, s\, (mkeps(r\backslash s))) \\ & & \textit{else None} \end{array}$$

Proof. By routine induction on s and generalisation over r. We need to use auxiliary properties about *flex* such as

$$g\,(\textit{flex}\ r\ f\ s\ v) = \textit{flex}\ r\ (g \circ f)\ s\ v$$

which can be easily established by induction on s. □

Note we did not redefine *lexer*, we just established that the value generated by *lexer* can also be obtained by a different method. While this different method is not efficient (we essentially need to traverse the string s twice, once for building the derivative $r\backslash s$ and another time for stacking up injection functions using *flex*), it will help us later with proving that incrementally building up values as done in Sulzmann and Lu's bitcoded version of the lexing algorithm is correct.

For convenience we use the following simple Isabelle/HOL datatype for representing bit-sequences (list of *bit*s).

$$\textit{bit} ::= Z \mid S$$

The coding function for translating values into bit-sequences is relatively straightforward.

Definition 13 (Bitcoding of Values)**.**

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

As can be seen, this coding is "lossy" in the sense that we do not record explicitly character values and also not sequence values (for them we just append two bit-sequences). We do, however, record the different alternatives for $\textit{Left}$, respectively $\textit{Right}$, as Z and S followed by some bit-sequence. Similarly, we use Z to indicate if there is still a value coming in the list of $\textit{Stars}$, whereas S indicates the end of the list. The lossiness makes the process of decoding a bit more involved, but the point is that if we have a regular expression *and* a bit-sequence of a corresponding value, then we can always decode the value accurately. The decoding can be defined by using two functions called *decode′* and *decode*:

Definition 14 (Bitdecoding of Values)**.**

$$
\begin{array}{ll}
\textit{decode}'\,bs\,(\mathbf{1}) & \stackrel{\text{def}}{=} (\textit{Empty}, bs) \\
\textit{decode}'\,bs\,(c) & \stackrel{\text{def}}{=} (\textit{Char}\,c, bs) \\
\textit{decode}'\,(Z :: bs)\,(r_1 + r_2) & \stackrel{\text{def}}{=} \textit{let}\,(v, bs_1) = \textit{decode}'\,bs\,r_1\;\textit{in}\;(\textit{Left}\,v, bs_1) \\
\textit{decode}'\,(S :: bs)\,(r_1 + r_2) & \stackrel{\text{def}}{=} \textit{let}\,(v, bs_1) = \textit{decode}'\,bs\,r_2\;\textit{in}\;(\textit{Right}\,v, bs_1) \\
\textit{decode}'\,bs\,(r_1 \cdot r_2) & \stackrel{\text{def}}{=} \textit{let}\,(v_1, bs_1) = \textit{decode}'\,bs\,r_1\;\textit{in}
\end{array}
$$

$$\begin{array}{lll}
 & & \textit{let}\,(v_2, bs_2) = \textit{decode}'\, bs_1\, r_2 \\
 & & \quad \textit{in}\;(\textit{Seq}\, v_1\, v_2, bs_2) \\
\textit{decode}'\,(Z :: bs)\,(r^*) & \stackrel{\text{def}}{=} & (\textit{Stars}\,[], bs) \\
\textit{decode}'\,(S :: bs)\,(r^*) & \stackrel{\text{def}}{=} & \textit{let}\,(v, bs_1) = \textit{decode}'\, bs\, r\;\textit{in} \\
 & & \textit{let}\,(\textit{Stars}\, vs, bs_2) = \textit{decode}'\, bs_1\, r^* \\
 & & \quad \textit{in}\;(\textit{Stars}\, v :: vs, bs_2) \\
 & & \\
\textit{decode}\, bs\, r & \stackrel{\text{def}}{=} & \textit{let}\,(v, bs') = \textit{decode}'\, bs\, r\;\textit{in} \\
 & & \textit{if}\; bs' = []\;\textit{then Some}\; v\;\textit{else None}
\end{array}$$

Note, we can only detect from the **1** regular expressions, respectively char regular expression, that an $Empty$ or, respectively, a character value needs to be generated. This is because the empty string and characters are not encoded into the bit-sequence. Since there is no “in-between marker” for when two values have to be calculated for a sequence, decoding needs to thread the bit-sequence through different calls (see clauses for sequence and star regular expressions). This means that *decode*$'$ attempts to “nibble” off parts of the bit-sequence according to the shape of the regular expression and leaves something as left-over bit-sequence that has not yet been decoded.

The function *decode*$'$ is well-defined because either the size of the regular expression decreases in each call, or if not, then the bit-sequence gets shorter. In the main decoding function *decode* we explicitly record that decoding can fail, producing *None*. For example in instances where the “left-over” bit-sequence is not completely consumed. This can be the case when the bit-sequence does not correspond to the regular expression with which *decode* is called. We can establish that for a value v inhabited by a regular expression r, the decoding of its bit-sequence never fails.

Lemma 13. *If* $\vdash v : r$ *then* $\textit{decode}\,(\textit{code}\,v)\,r = \textit{Some}\,v$.

Proof. This follows from the property that $\textit{decode}'\,((\textit{code}\,v)\,@\,bs)\,r \;=\; (v, bs)$ holds for any bit-sequence bs and $\vdash v : r$. This property can by easily proved by induction on $\vdash v : r$. $\square$

Sulzmann and Lu also introduce *annotated regular expressions*, which are the usual regular expressions plus an extra bit-sequence annotated to each constructor (except **0**). Annotated regular expressions are the main data structure over which the bitcoded algorithm works and which can be defined as an Isabelle/HOL datatype as follows:

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

where bs are bit-sequences and the as are annotated regular expressions. In what follows, we shall use the convention that r will stand for "standard" regular expressions, and a for annotated regular expressions.

Sulzmann and Lu define the function *internalise* in order to transform standard regular expressions into annotated regular expressions. We write this operation as $r^{\uparrow}$. This internalisation uses the following *fuse* function.

Definition 15 (Fuse Function)**.**

$$\begin{array}{lcl}
\textit{fuse}\,bs\,(\textit{ZERO}) & \stackrel{\text{def}}{=} & \textit{ZERO} \\
\textit{fuse}\,bs\,(\textit{ONE}\,bs') & \stackrel{\text{def}}{=} & \textit{ONE}\,(bs\,@\,bs') \\
\textit{fuse}\,bs\,(\textit{CHAR}\,bs'\,c) & \stackrel{\text{def}}{=} & \textit{CHAR}\,(bs\,@\,bs')\,c
\end{array}$$

$$\begin{array}{lcl}
\textit{fuse}\, bs\, (\textit{ALT}\, bs'\, a_1\, a_2) & \stackrel{\text{def}}{=} & \textit{ALT}\, (bs\, @\, bs')\, a_1\, a_2 \\
\textit{fuse}\, bs\, (\textit{SEQ}\, bs'\, a_1\, a_2) & \stackrel{\text{def}}{=} & \textit{SEQ}\, (bs\, @\, bs')\, a_1\, a_2 \\
\textit{fuse}\, bs\, (\textit{STAR}\, bs'\, a) & \stackrel{\text{def}}{=} & \textit{STAR}\, (bs\, @\, bs')\, a
\end{array}$$

Definition 16 (Internalisation)**.**

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

There is also an *erase*-function, written $a^\downarrow$, which transforms an annotated regular expression into a (standard) regular expression by just erasing the annotated bit-sequences. We omit the straightforward definition. For defining the algorithm, we also need the functions *bnullable* and *bmkeps*, which are the "lifted" versions of *nullable* and *mkeps* acting on annotated regular expressions, instead of regular expressions.

Definition 17 (*bnullable*)**.**

$$\begin{array}{lcl}
\textit{bnullable}\, (\textit{ZERO}) & \stackrel{\text{def}}{=} & \textit{false} \\
\textit{bnullable}\, (\textit{ONE}\, bs) & \stackrel{\text{def}}{=} & \textit{true} \\
\textit{bnullable}\, (\textit{CHAR}\, bs\, c) & \stackrel{\text{def}}{=} & \textit{false} \\
\textit{bnullable}\, (\textit{ALT}\, bs\, a_1\, a_2) & \stackrel{\text{def}}{=} & \textit{bnullable}\, a_1 \vee \textit{bnullable}\, a_2 \\
\textit{bnullable}\, (\textit{SEQ}\, bs\, a_1\, a_2) & \stackrel{\text{def}}{=} & \textit{bnullable}\, a_1 \wedge \textit{bnullable}\, a_2 \\
\textit{bnullable}\, (\textit{STAR}\, bs\, a) & \stackrel{\text{def}}{=} & \textit{true}
\end{array}$$

Definition 18 (*bmkeps*)**.**

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

The key function in the bitcoded algorithm is the derivative of an annotated regular expression. This derivative calculates the derivative but at the same time also the incremental part that contributes to constructing a value.

Definition 19 (Derivative of Annotated Regular Expressions)**.**

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

This function can also be extended to strings, written $a\backslash s$, just like the standard derivative. We omit the details. Finally we can define Sulzmann and Lu's bitcoded lexer, which we call *blexer*:

Definition 20.

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

This bitcoded lexer first internalises the regular expression r and then builds the annotated derivative according to s. If the derivative is nullable, then it extracts the bitcoded value using the *bmkeps* function. Finally it decodes the bitcoded value. If the derivative is *not* nullable, then *None* is returned. The task is to show that this way of calculating a value generates the same result as with *lexer*.

Before we can proceed we need to define a function, called *retrieve*, which Sulzmann and Lu introduced for helping with the proof argument.

Definition 21 (*Retrieve*)**.**

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

The idea behind this function is to retrieve a possibly partial bitcode from an annotated regular expression, where the retrieval is guided by a value. For example if the value is $\textit{Left}$ then we descend into the left-hand side of an alternative (annotated) regular expression in order to assemble the bitcode. Similarly for $\textit{Right}$. The property we can show is that for a given v and r with $\vdash v : r$,

the retrieved bitsequence from the internalised regular expression is equal to the bitcoded version of v.

Lemma 14. If $\vdash v : r$ then $\textit{code}\, v = \textit{retrieve}\,(r^{\uparrow})\, v$.

Proof. By induction on $\vdash v : r$. There are no interesting cases. □

We also need some auxiliary facts about how the bitcoded operations relate to the "standard" operations on regular expressions. For example if we build a bitcoded derivative and erase the result, this is the same as if we first erase the annotated regular expression and then perform the "standard" derivative operation.

Lemma 15.

(1) $(a\backslash s)^{\downarrow} = (a^{\downarrow})\backslash s$

(2) $\textit{bnullable}(a)$ iff $\textit{nullable}(a^{\downarrow})$

(3) $\textit{bmkeps}(a) = \textit{retrieve}\, a\, (\textit{mkeps}\,(a^{\downarrow}))$ provided $\textit{nullable}(a^{\downarrow})$.

Proof. All properties are by induction on annotated regular expressions. There are no interesting cases. □

This brings us to our main lemma in this section: if we build a derivative, say $r\backslash s$ and have a value, say v, inhabited by this derivative, then we can produce the result $lexer$ generates by applying this value to the stacked-up injection functions *flex* assembles. The lemma establishes that this is the same value as if we build the annotated derivative $r^{\uparrow}\backslash s$ and then retrieve the corresponding bitcoded version, followed by a decoding step.

Lemma 16 (Main Lemma)**.** *If* $\vdash v : r\backslash s$ *then*

$$\textit{Some}\,(\textit{flex}\, r\, \textit{id}\, s\, v) = \textit{decode}(\textit{retrieve}\,(r^{\uparrow}\backslash s)\, v)\, r$$

Proof. This can be proved by induction on s and generalising over v. The interesting point is that we need to prove this in the reverse direction for s. This

means instead of cases [] and $c :: s$, we have cases [] and $s\,@\,[c]$ where we unravel the string from the back.[3]

The case for [] is routine using Lemmas 13 and 14. In the case $s\,@\,[c]$, we can infer from the assumption that $\vdash v : (r\backslash s)\backslash c$ holds. Hence by Lemma 6 we know that (*) $\vdash inj\ (r\backslash s)\ c\ v : r\backslash s$ holds too. By definition of *flex* we can unfold the left-hand side to be

$$\textit{Some}\,(\textit{flex}\ r\ \textit{id}\,(s\,@\,[c])\,v) = \textit{Some}\,(\textit{flex}\ r\ \textit{id}\,s\,(inj\ (r\backslash s)\,c\,v))$$

By induction hypothesis and (*) we can rewrite the right-hand side to

$$\textit{decode}\,(\textit{retrieve}\,(r^{\uparrow}\backslash s)\ (inj\ (r\backslash s)\,c\ v))\,r$$

which is equal to $\textit{decode}\,(\textit{retrieve}\,(r^{\uparrow}\backslash(s\,@\,[c]))\,v)\,r$ as required. The last rewrite step is possible because we generalised over v in our induction. □

With this lemma in place, we can prove the correctness of *blexer* such that it produces the same result as *lexer*.

Theorem 6. $\textit{lexer}\,r\ s = \textit{blexer}\,r\ s$

Proof. We can first expand both sides using Lemma 12 and the definition of *blexer*. This gives us two *if*-statements, which we need to show to be equal. By Lemma 15*(2)* we know the *if*-tests coincide:

$$\textit{bnullable}(r^{\uparrow}\backslash s)\ \textit{iff}\ nullable(r\backslash s)$$

For the *if*-branch suppose $a \stackrel{\text{def}}{=} r^{\uparrow}\backslash s$ and $d \stackrel{\text{def}}{=} r\backslash s$. We have (*) $nullable\,d$. We

[3]Isabelle/HOL provides an induction principle for this way of performing the induction.

can then show by Lemma 15*(3)* that

$$\textit{decode}(\textit{bmkeps}\, a)\, r = \textit{decode}(\textit{retrieve}\, a\, (\textit{mkeps}\, d))\, r$$

where the right-hand side is equal to $\textit{Some}\,(\textit{flex}\, r\, \textit{id}\, s\, (\textit{mkeps}\, d))$ by Lemma 16 (we know $\vdash \textit{mkeps}\, d : d$ by (*)). This shows the *if*-branches return the same value. In the *else*-branches both *lexer* and *blexer* return *None*. Therefore we can conclude the proof. □

To sum up, we have established that the bitcoded algorithm by Sulzmann and Lu without simplification produces correct results. This was only conjectured in their paper [53]. 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.

3.3 Extensions

A strong point in favour of Sulzmann and Lu's algorithm is that it can be extended in various ways. If we are interested in tokenising a string, then we need to not just split up the string into tokens, but also "classify" the tokens (for example whether it is a keyword or an identifier). This can be done with only minor modifications to the algorithms by introducing *record regular expressions* and *record values* (see for example [54]). For this recall our definitions of regular expressions and values on Pages 30 and 37 and extend them as follows:

$$r := \ldots \mid (l : r) \qquad v := \ldots \mid (l : v)$$

where l is a label, say s a string, r a regular expression and v a value. All functions can be smoothly extended to this additional regular expression and value.

For example $(l : r)$ is nullable iff r is; the derivative, that is $(l : r)\backslash c$, is defined as $(l : r\backslash c)$ and so on. The purpose of the record regular expression is to mark certain parts of a regular expression and then record in the calculated value which parts of the strings were matched by this part. The label can then serve as classification for the tokens. For this recall the regular expression $(r_{key} + r_{id})^*$ for keywords and identifiers from the Introduction. With the record regular expression we can form $((key : r_{key}) + (id : r_{id}))^*$ and then traverse the calculated value and only collect the underlying strings in record values. With this we obtain finite sequences of pairs of labels and strings, for example

$$(l_1 : s_1), ..., (l_n : s_n)$$

from which tokens with classifications (keyword-token, identifier-token and so on) can be extracted. One way to do this is to traverse a value and collect all flattened strings of marked subvalues and associate them with the labels. This can be defined as follows:

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

where *concat* "flattens" a list of lists to just a single list and where $|v|$ produces the underlying string of a value. This is how we envisage a lexer can be implemented based on Sulzmann and Lu's algorithm.

In the context of POSIX matching, it is also interesting to study additional constructors about *bounded repetitions* of regular expressions. For this let us

extend the results from the previous sections to the following four additional regular expression constructors:

$$\begin{array}{lll} r := \ldots & \mid r^{\{n\}} & \text{exactly-}n\text{-times} \\ & \mid r^{\{..n\}} & \text{upto-}n\text{-times} \\ & \mid r^{\{n..\}} & \text{from-}n\text{-times} \\ & \mid r^{\{n..m\}} & \text{between-}nm\text{-times} \end{array}$$

In what follows we shall call them *bounded regular expressions*. With the help of the power operator (definition omitted) on languages, the associated languages recognised by these regular expressions can be defined in Isabelle as follows:

$$\begin{array}{lll} L(r^{\{n\}}) & \stackrel{\text{def}}{=} & L(r)^n \\ L(r^{\{..n\}}) & \stackrel{\text{def}}{=} & \bigcup_{i\in\{..n\}} .\ L(r)^i \\ L(r^{\{n..\}}) & \stackrel{\text{def}}{=} & \bigcup_{i\in\{n..\}} .\ L(r)^i \\ L(r^{\{n..m\}}) & \stackrel{\text{def}}{=} & \bigcup_{i\in\{n..m\}} .\ L(r)^i \end{array}$$

This definition uses of the convenient interval definitions in Isabelle/HOL. For example $\{n..m\}$ stands for the interval $n \leq i \leq m$; similarly $\{..n\}$ stands for $0 \leq i \leq n$ and so on. The definition in Isabelle/HOL implies that in the last clause $r^{\{n..m\}}$ matches no string if $m < n$, because then the interval $\{n..m\}$ is empty.

Note that we are a bit over-generous in our use of primitives: for example exactly-n-times $r^{\{n\}}$ could be substituted with $r^{\{n..n\}}$; similarly upto-n-times $r^{\{..n\}}$ could be substituted with $r^{\{0..n\}}$. We could even drop the Kleene star by substituting r^* with $r^{\{0..\}}$. But for the sake of argument, let us explain the details for all bounded repetition constructors.

While the language recognised by these regular expressions is straightforward, some care is needed when defining the corresponding lexical values. First with a slight abuse of language, we will (re)use values of the form *Stars* vs for

values inhabited by bounded regular expressions.[4] Second, we need to introduce rules for extending our inhabitation relation given in Definition 6 on Page 38, from which we then derived our notion of lexical values, LV. Given the rule for r^*, the rule for $r^{\{..n\}}$ is relatively straightforward: it just requires additionally that the length of the list of values must be smaller or equal to n, that is

$$\frac{\forall v \in vs.\ \vdash v : r \wedge |v| \neq [] \qquad \textit{len}\ vs \leq n}{\vdash \textit{Stars}\ vs : r^{\{..n\}}}$$

Like in the r^*-rule, we ensure with the left-premise that some non-empty part of the string is "chipped" away by *every* value in vs, that means the corresponding values do not flatten to the empty string.

Matters are bit more complicated in the rule for $r^{\{n\}}$ (that is exactly n-times r). We clearly need to require that the length of the list of values equals to n. But requiring that every of these n values "chips" away some non-empty part of a string would be too strong. According to the informal POSIX rules we have to allow that there is an "initial segment" that needs to chip away some parts of the string, but if this segment is too short for satisfying the exactly-n-times constraint, it can be followed by a segment where every value flattens to the empty string. We found that the only way for expressing this constraint in Isabelle is by rules of the form:

$$\frac{\begin{array}{l}\forall v \in vs_1.\ \vdash v : r \wedge |v| \neq [] \\ \forall v \in vs_2.\ \vdash v : r \wedge |v| = [] \\ \textit{len}\ (vs_1\ @\ vs_2) = n\end{array}}{\vdash \textit{Stars}\ (vs_1\ @\ vs_2) : r^{\{n\}}}$$

The vs_1 is the initial segment with non-empty flattened values, whereas vs_2 is the segment where all values flatten to the empty string. This idea gets even

[4] The alternative would be to introduce a separate constructor, for example *List* vs. But this seems overkill given the relatively little benefit from such a naming scheme.

more complicated for the $r^{\{n..\}}$ regular expression. The reason is that we need to distinguish the case where we use fewer repetitions than n. In this case we need to "fill" the end with values that match the empty string to obtain at least n repetitions. But in case we need more than n repetitions, then *all* values should match a non-empty string. This leads to two rules:

$$\frac{\forall v \in vs_1.\ \vdash v : r \wedge |v| \neq [] \quad \forall v \in vs_2.\ \vdash v : r \wedge |v| = [] \quad \mathit{len}\ (vs_1 \mathbin{@} vs_2) = n}{\vdash \mathit{Stars}\ (vs_1 \mathbin{@} vs_2) : r^{\{n..\}}} \qquad \frac{\forall v \in vs.\ \vdash v : r \wedge |v| \neq [] \quad \mathit{len}\, vs > n}{\vdash \mathit{Stars}\ vs : r^{\{n..\}}}$$

Note that these two rules "collapse" in case $n = 0$ to just the single rule given for r^* in Definition 6. We have similar rules for the between-nm-times operator. These rules ensure that our definition for sets lexical values $LV\ r\ s$ is still finite and also fits with the ordering given by Okui and Suzuki (which require minimal values over the sets $LV\ r\ s$).

Fortunately, the other definition extend "smoother" to bounded repetitions. For example the rules for derivatives are:

$$\begin{array}{lcl}
r^{\{n\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0\ \mathit{then}\ \mathbf{0}\ \mathit{else}\ (r\backslash c) \cdot r^{\{n-1\}} \\
r^{\{..n\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0\ \mathit{then}\ \mathbf{0}\ \mathit{else}\ (r\backslash c) \cdot r^{\{..n-1\}} \\
r^{\{n..\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ n = 0\ \mathit{then}\ (r\backslash c) \cdot r^*\ \mathit{else}\ (r\backslash c) \cdot r^{\{n-1..\}} \\
r^{\{n..m\}}\backslash c & \stackrel{\text{def}}{=} & \mathit{if}\ m < n\ \mathit{then}\ \mathbf{0} \\
 & & \mathit{else}\ \mathit{if}\ n = 0\ \mathit{then} \\
 & & \quad\quad \mathit{if}\ m = 0\ \mathit{then}\ \mathbf{0}\ \mathit{else}\ (r\backslash c) \cdot r^{\{..m-1\}} \\
 & & \mathit{else}\ (r\backslash c) \cdot r^{\{n-1..m-1\}}
\end{array}$$

For *mkeps* we need to generate the shortest list of values we can get "away with". This means for example in the case $r^{\{..n\}}$ we can return the empty list, like for stars. In the other cases we have to generate a list of exactly n copies of the *mkeps*-value, because n is the smallest number of repetitions.

$$\begin{array}{lcl}
\textit{mkeps}\ (r^{\{..n\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ []\\
\textit{mkeps}\ (r^{\{n\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{replicate}\ n\ (\textit{mkeps}\ r))\\
\textit{mkeps}\ (r^{\{n..\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{replicate}\ n\ (\textit{mkeps}\ r))\\
\textit{mkeps}\ (r^{\{n..m\}}) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{replicate}\ n\ (\textit{mkeps}\ r))
\end{array}$$

In this definition we use Isabelle's *replicate*-function in order to generate a list of n copies of a value. The injection function also extends straightforwardly to the bounded regular expressions as follows:

$$\begin{array}{lcl}
\textit{inj}\ (r^{\{n\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ vs)) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: vs)\\
\textit{inj}\ (r^{\{n..\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ vs)) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: vs)\\
\textit{inj}\ (r^{\{..n\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ vs)) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: vs)\\
\textit{inj}\ (r^{\{n..m\}})\ c\ (\textit{Seq}\ v\ (\textit{Stars}\ vs)) & \stackrel{\text{def}}{=} & \textit{Stars}\ (\textit{inj}\ r\ c\ v :: vs)
\end{array}$$

Similarly our POSIX definition can be easily extended to the additional constructors. For example for $r^{\{n\}}$ we have two rules:

$$\frac{\forall v \in vs.\ ([], r) \to v \qquad \textit{len}\ vs = n}{([], r^{\{n\}}) \to Stars\ vs}$$

$$\frac{\begin{array}{c}(s_1,\ r) \to v \qquad (s_2,\ r^{\{n-1\}}) \to\ Stars\ vs \qquad |v| \neq [] \qquad 0 < n\\ \nexists s_3\ s_4.\ \ s_3 \neq []\ \wedge s_3 @ s_4 = s_2 \wedge s_1 @ s_3 \in L(r) \wedge s_4 \in L(r^{\{n-1\}})\end{array}}{(s_1 @ s_2,\ r^{\{n\}}) \to Stars\ (v :: vs)}$$

The first rule deals with the case when an empty string needs to be recognised. The second when the string is non-empty. In this case the "initial segment" must match non-empty strings only. The idea behind this formulation is to avoid situations where an earlier value matches the empty string, while it is actually possible to "nibble away" some parts of the string. The rules for the other bounded regular expressions are similar. We shall omit them. With these definitions in place, our proofs given in the previous sections extend to the bounded repetitions. The main point is that there are no surprises.

Unfortunately, in our formal proofs we need to give the proofs all over again in a separate theory, since there is no way of making Isabelle to accept proofs for the basic regular expressions (defined as inductive datatype) and then augmenting the datatype with new constructors. This would be a really "cool" feature for Isabelle, but we have no idea how this could be achieve without too much effort. Unfortunately, what is also *not* known is what a "complete" set of simplification rules should be for bounded repetitions. It seems with the derivatives we have given earlier, the regular expressions are prone to a linear growth in size. It seems what is needed is that bounded repetition need to be "compactified". For example if one has instances of $r^{\{n\}}$ and $r^{\{n-1\}}$ inside an alternative regular expression—in such cases, it seems, we need to replace them with $r^{\{n-1..n\}}$ in order to keep the size of derivatives "small" and thus make the algorithm efficient.

3.4 Summary and Future Work

We set out in this work to implement in Isabelle/HOL the lexing algorithm by Sulzmann and Lu and formalise the extensive "pencil-and-paper" notes given by them for establishing the correctness of their algorithm [53]. In our opinion, the extension of Brzozowski's matching algorithm by a second phase that generates a POSIX value for how a regular expression matched a string is really clever and beautiful. The hope was that the formalisation of the extension would be similarly simple as the formalisation of the correctness for Brzozowski's regular expression matching algorithm from 1964 [15]. We were therefore rather surprised, even dumbstruck, that no matter what we attempted, the arguments Sulzmann and Lu did not enable us to make any progress with a formalisation in Isabelle/HOL. We have (on and off) explored mechanisations as soon as first versions of [53] appeared, but made little or no progress with turning the relatively

detailed proof sketches in [53] into a formalised proof. Having seen the work by Vansummeren [58] and adapting his POSIX definition for the algorithm by Sulzmann and Lu made all the difference: the proofs then were nearly straightforward. We also appreciate very much the work by Okui and Suzuki [41] which allowed us to gain more confidence that our definition really captures the "spirit" of the informal POSIX rules. There are also alternative definitions which capture the informal rules in a distinct way, for example [8]. Our Isabelle/HOL code is available from

`https://github.com/fahadausaf/POSIX-Parsing`

The results from Chapter 1, as well as from Sections 2.1 and 3.1 are also in the Archive of Formal Proofs of Isabelle.[5]

Having proved the correctness of the POSIX lexing algorithm in [53], which lessons have we learned? Well, we feel this is a perfect example for the importance of the right definitions and formalised proofs. Our proofs were done both done by hand and checked in Isabelle/HOL. The experience of doing our proofs in this way has been that the mechanical checking was absolutely essential: despite the apparent maturity, this subject area has hidden snares. If we had only relied on "pencil-and-paper" proofs we would have also been overwhelmed with faulty reasoning—in particular in one instance, only the formalisation saved us from serious errors and wrong statements.

There are many avenues for future research. The most ambitious research goal would be, in our opinion, to make progress with back-references and derivative based regular expression matching. Back-references seem indispensable in application, such as Snort and Bro. While the overall (matching) problem is then NP-complete, users who employ back-references are clearly not interested in the

[5] `https://www.isa-afp.org/entries/Posix-Lexing.html`

full generality of the problem. Rather they seem to be interested in subproblems that can be solved efficiently. Alas nothing is known about how to restrict the problem using the derivatives approach, or any other approach for that matter. Moreover, given the example by Aho [3] for establishing the NP-completeness, it seems challenging to make any progress soon.

More feasible seems to be to make progress with the bitcoded algorithm and finding a set of simplification rules that keep the sizes of derivatives small (with some appropriate definition what "small" means). Our conjecture is that one needs to mimic with simplification rules the partial derivatives for regular expression introduced by Antimirov [5]. He established an upper bound for how much partial derivatives can grow in terms of the size of the regular expression, but is independent from the length of strings. The idea would be to adapt his bound to the case of "standard" derivatives by having rather "aggressive" simplification rules. The ones described in Section 3.1 are clearly not aggressive enough in order to obtain such an upper bound.

However, there seem to be also a number of rather low-hanging fruits that can be investigated in order to make the Sulzmann and Lu algorithm faster. For example Murugesan and Shanmuga Sundaram describe an idea that the usual derivative operation, which iterates the derivative character-by-character, can be defined bigger "chunks" [40]. For example if we want to calculate $(r_1 + r_2)\backslash s$ we can immediately replace this by $(r_1\backslash s) + (r_2\backslash s)$ rather that having to iterate the derivative character-by-character, as in Definition 5, in order to obtain the same result. They also give some details about how to do this for sequence and star regular expressions, but whether they improve efficiency remains to be seen. However, it would be interesting to see if such an idea can also be made to work with the injection function by Sulzmann and Lu, which also just iterates the injection character-by-character (at least for alternative regular expressions).

Alas due to time constraints we were not able to consider this. Similarly we were not able to fully work out the details of the *not*-derivative from [43]. This should be relatively straightforward (the only real change would be to have to define the inhabitation relation as recursive function, rather than as inductive predicate).

Another area of interest is to short-circuit the lexing algorithm outlined in Section 3.3 using record regular expressions and record values. Since in a lexer one is only interested in the token sequence, it seems overkill to calculate the complete value first and then extract the token sequence from the calculated value. In our opinion there must be a way to calculate the token sequence more directly without the detour of calculating the value first. We leave all these questions as further work.

Part II

TLS Message Parsers

Chapter 4

Project Everest

This part is about verified implementations of different parsers for parsing messages in TLS—the *Transport Layer Security Protocol*. This protocol is the security layer of HTTPS. These parsers are implemented as part of the *Everest Project*,[1] which is an umbrella project for producing verified implementations for different components of HTTPS. It includes, for example, a verified reference implementation of TLS, called miTLS,[2] and verified implementations of different cryptographic algorithms, such as AES, SHA2 and so on [12, 13, 21]. The Everest Project is implemented in the F* language,[3] which is an ML-like functional programming language aimed at program verification. Code in F* can be extracted to F♯ and OCaml.

The work reported here arose from an internship at Microsoft Research in Cambridge in 2017. The task was to implement a new and sound bytes library in F*, which forms part of the basic infrastructure for the Project Everest. Then existing (verified) parsers needed to be ported to this library.

[1] https://project-everest.github.io
[2] https://mitls.org
[3] https://www.fstar-lang.org

4.1 Introduction

A problem with F♯ and OCaml is that their runtime environments needs to perform garbage collection. This often slows down programs and running times may become unpredictable [37], which is not acceptable for many security-related applications such as TLS. Another problem is that these runtime environments are not easy to integrate with (mainstream) applications such as Skype, Internet Explorer and IIS, which are mostly written in C and use the low-level runtime environment of C. Therefore, in order to have a more predictable, and fast and easy to integrate runtime environment, the Everest team decided to migrate the software in the Everest Project from the functional languages OCaml and F♯ to the low-level language C.

As part of this migration, our focus is here on the specification of TLS parsers. We have established correctness and security properties for parser specifications and then derived efficient and composable implementations from these specifications. These implementations are extracted to C using the recent tactics engine of F*. For this, we created a library providing a unified model for bytes, replacing the previous unsound library. We then updated the TLS parsers to use this new library and enhanced their functionality and improved the verification automation. To see how this work fits together, let us first describe briefly the HTTPS ecosystem and the Everest Project.

4.2 The HTTPS Ecosystem

HTTPS and TLS are key protocols on which almost all of the Internet is built upon. TLS is the main security protocol inside HTTPS. It consists of a *protocol layer* and a *record layer*. The protocol layer is used for establishing a secure connection when communicating with a third party. It negotiates some crypto-

graphic parameters, algorithms and cipher suites. The record layer is concerned with the encryption of the transmitted data.

QUIC—the *Quick UDP Internet Connection* [17, 35] is another, more recent Internet security protocol developed by Google. It uses TLS handshakes to negotiate parameters, to agree upon cipher algorithms and to perform key exchanges. On top of it, it supports multiplexed streams between endpoints over UDP—the *User Datagram Protocol*. This reduces the number of connections between endpoints and therefore reduces latency. If QUIC features can be proved to be correct, then it is likely that it becomes part of later version of TLS.

In Figure 4.1 is a rough overview over the HTTPS ecosystem, with the unsecured network on the bottom and services and applications on the top. TLS is part of HTTPS providing security protocols. Unfortunately, all these protocols are very brittle in terms of security. Attacks, such as LogJam and FREAK [52, 14], frequently grab headlines. It is known that the implementations of these protocols contain bugs, especially in their core cryptographic algorithms. The bugs not only affect the functional correctness, but also affect side-channel resistance. Often the reason for these bugs lies with optimisations in the core cryptographic algorithms, which are critical for speed and performance. For example, there are a number of known bugs in Open-SSL [10, 36], where engineers implemented very “hardcore” optimisations for their cryptographic algorithms and as a result these implementations either produce the wrong result or generate very annoying and dangerous buffer overflows.

Verification of this ecosystem is often difficult, especially when one combines cryptographic algorithms in order to obtain high-level abstractions for authentication and encryption. This is where you reason about the combination of these algorithms in order to obtain an “upper-bound” on the strength of the cryptographic algorithms and prove that the cryptographic abstractions preserves

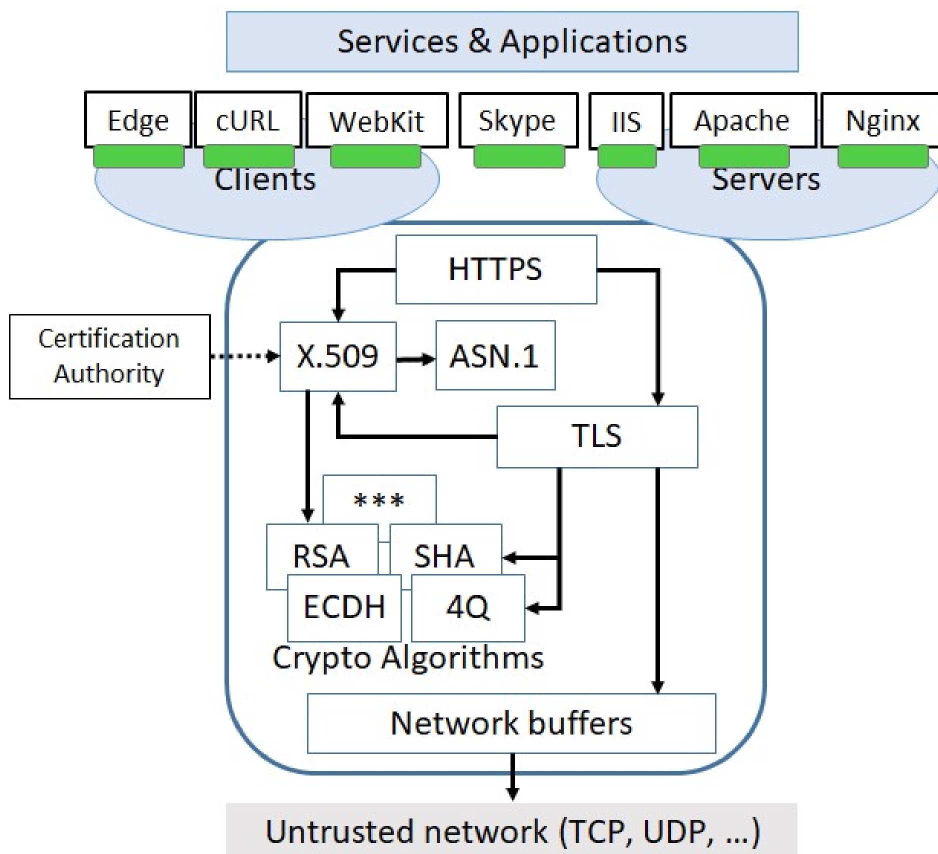


Figure 4.1: A rough overview over the HTTPS ecosystem given by the Everest Project.

integrity, authenticity and confidentiality.

Handshakes are another very important aspect of the HTTPS ecosystem, which are often based on state machines. Their purpose is to exchange messages and depending on what servers answer, clients may or may not be allowed to proceed with a key exchange or some other set of algorithms. These handshakes are notorious places for habouring bugs [39]. The reason is that engineers have made handshakes more and more complex because they want to start transmitting data as soon as possible. This compromises however often on the guarantees one expects from these handshakes, such as forward secrecy.

Parsing messages is another "minefield" in the HTTPS ecosystem. Messages might come from the unsecured network and we need to write parsers for them

that do not trigger buffer overflows or other errors. And finally, if one wants to implement a protocol, such as QUIC, one ends up in the realm of low-level programming where one has to manipulate low-level data structures. There concurrency can bite and also interaction with the operating system are necessary, which are far from straightforward to implement and to reason about. Because of all these difficulties and problems, the Everest Project has been founded in order to address the challanges in a formal and verified fashion.

4.3 Project Everest

The Everest acronym stands for **E**xpedition for a **Ver**ified **S**ecure **T**ransport. It is an umbrella project for developing verified implementation of different components of HTTPS protocol. This includes the TLS protocol as the main security protocol of HTTPS. The Everest Project already includes verification of different underlying cryptographic algorithms and cipher suites such as AES, and SHA2 etc. The aim of the Everest is to develop a verified secure protocol library of all the important Internet protocols. For this it combines verification methods in order to obtain strong security guarantees, along with usability and practicality as we want the code to be usable in actual applications (for example Edge, WebKit, Skype and IIS). The project is developed using two sets of tools: The first is a verification tool called F*, an ML-like functional programming language aimed at program verification. F* is used for the implementation of most of the security protocols. The second is a set of mechanized tools such as Vale, Low* (pronounced low star) and Kremlin which are used to compile F* code to C-code.

4.3.1 The Everest Toolchain

Project Everest is a combination of the following modules:

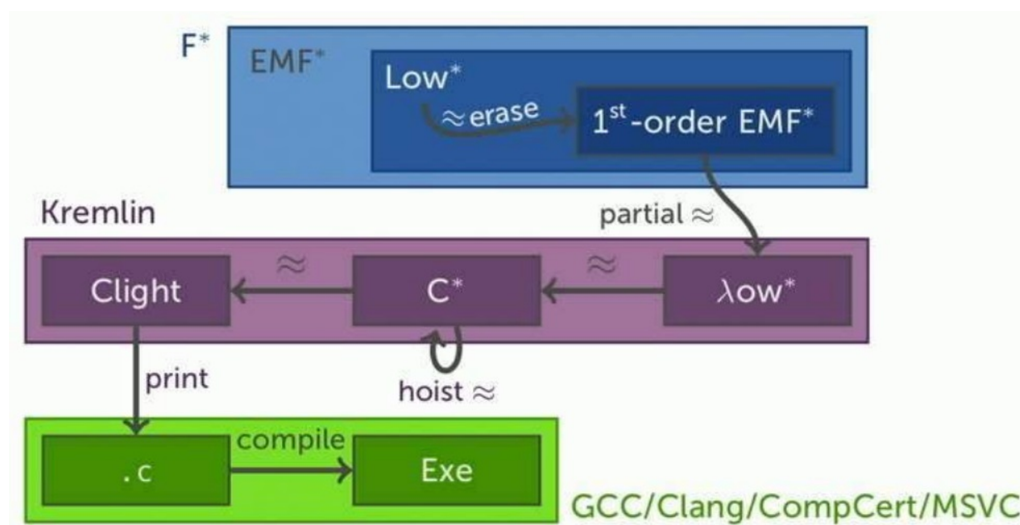


Figure 4.2: An overview over the Everest toolchain.

F⋆ Language

F⋆ is an ML-style functional programming language. Its type system includes polymorphism, dependent types, monadic effects and refinement types. Together these features allow users to write precise and compact specifications for programs, including the specification of functional correctness and security properties. The F⋆ type-checker attempts to prove that programs meet their specifications using a combination of SMT solving and manual proofs. The design goal of the language is to be user extensible.

MiTLS

MiTLS (*pronounced me-TLS*) is a verified reference implementation of the TLS protocol developed in collaboration with Microsoft Research and Inria. It interoperates with all the mainstream web servers and browsers and fully supports all the cipher-suites, wire formats, data fragmentations, sessions and connections, alerts and errors, re-handshakes and resumptions, and all the other requirements as prescribed in the RFCs. It is implemented entirely in F⋆ and the specifications

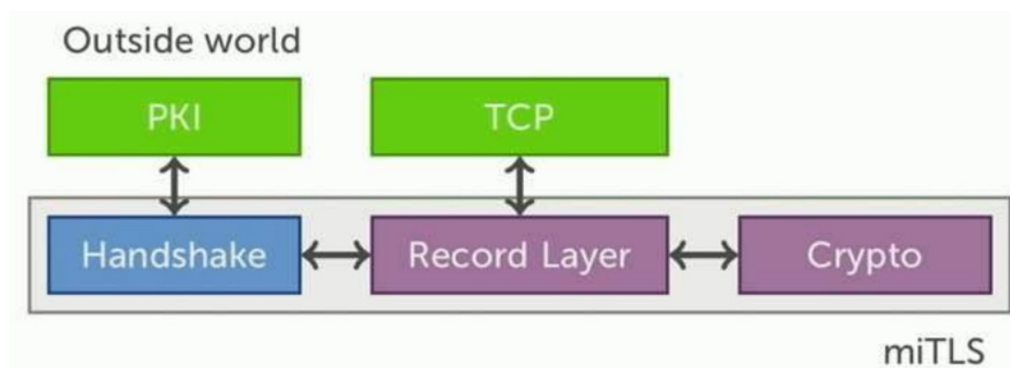


Figure 4.3: Overview over the reference implementation of TLS, called MiTLS.

are written in F7, which are refinement types for F*.

KreMLin

Everest currently extracts F* code to functional runtime based on OCaml and F#. KreMLin is developed to make it compatible with the low-level runtime of C and allow its integration with mainstream tools such as Chrome, Edge and IIS etc. It takes a subset of F* called Low-Star and produces C code.

HACL*

HACL stands for High-Assurance Cryptographic Library. It is a formally verified cryptographic library written entirely in F*. The goal of this library is to develop a verified C reference implementation for popular cryptographic primitives and to verify them for memory safety, functional correctness and secret independence.

Vale

Vale is a domain-specific language designed for emmitting formally verified high-performance assembly language code, with an emphasis on cryptographic code. It uses F* for formal verification and supports multiple architectures in-

cluding ARM, x86, x64, and multiple platforms including Linux, Windows and MacOSX.

4.3.2 The Everest Runtime

Project Everest is implemented using F* which compiles to two different runtime environments, namely a functional runtime environment which compiles to OCaml and F#, and a low-level runtime environment which compiles to C. The diagram below shows different components used in the functional and low-level runtime on the left and right side of the diagram respectively.

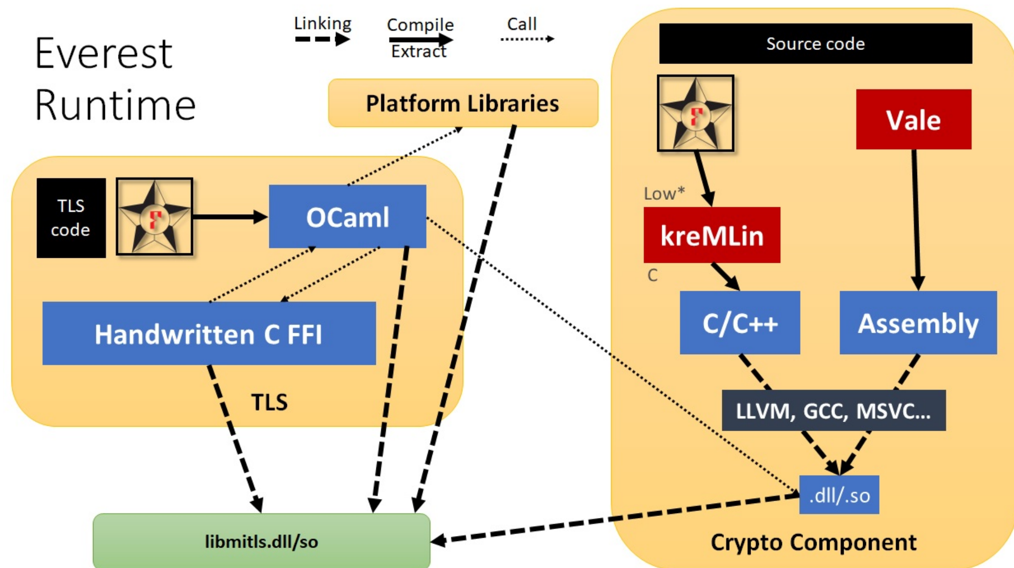


Figure 4.4: Everest runtime: left is the functional runtime and right is low-level runtime.

The low-level runtime is used for the implementation of the low-level cryptographic library (HACL) used in Project Everest. This runtime is implemented using a subset of F* called Low*, which is a shallow embedding of the C programming language. The Low* code is extracted to C using a tool called KreMLin. This low-level runtime environmen also provides support for assembly code which is compatible with the verified Low* code.

On the other hand, all of the Everest protocol code, particularly the TLS implementation, is currently written in the full F* language which cannot be extracted to C. The only way we could extract this code and use it with other applications is by extracting it to OCaml or F#. As mentioned before, the problem with the OCaml or F# code is the functional runtime which enforces garbage collection and adds an overhead which may affect the program performance. It is also difficult to integrate OCaml and F# with other mainstream applications. The only way we could integrate our verified TLS implementation with other applications is by using the *foreign function interface*[4] files which are handwritten in C and are unverified.

The Low-Level Runtime for Everest

One of the main reasons for migrating Everest from functional runtime environment to the low-level runtime is that we want to have a more predictable memory allocation and latency. The necessary garbage collection in functional runtime often results in high latency and the runtime becomes unpredictable. This leads to connections taking longer time and often information can also be leaked. For security related applications, it is very important to have low-level latency and more predictable runtime, specially in cryptographic algorithms where side-channels can be used for attacks.

Another important reason to migrate Everest to the low-level runtime of C is that it is much easier to integrate C with other applications. We will get rid of hand-written *foreign function interface* files which are unverified and potentially contain errors. We get better performance with C by using features which are not available in functional runtime environments such as machine integers. We also get rid of many other issue which are better addressed in C such as assembly

[4]Foreign function interface files are used to integrate Everest with other applications.

level support, performance on ARM architecture and other platforms. So naturally, in order to promote Everest so that we could easily integrate it with user applications, it is desirable that it compile to the industry-standard C code.

Chapter 5

A Pure Model of Bytes

Having described the F* language and the Everest Project, we shall describe in this chapter our new byte library. This library is part of the basic infrastructire for implementing TLS message parsers.

5.1 TLS Message Parsers

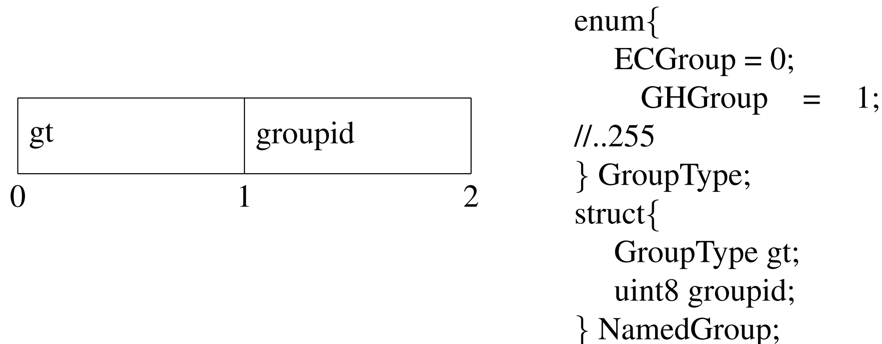


Figure 5.1: A simple TLS datatype structure.

Message parsing is the main entry point for all the data processing taking place inside the TLS protocol. Most of the data structures used in our TLS protocol implementation are given by the standard format of TLS messages [28]. Therefore, porting TLS message parsers to C is the first step towards porting all the

TLS to C. Figure 5.1 shows a simple TLS datatype structure.

This figure is an example for *named groups* defined in the TLS specifications. It is a struct containing two fields, a *GroupType* which is encoded with one byte, and a *GroupId*. The network encoding for this datatype is the concatenation of encoding of each element. An interesting point we have in this abstract message structure as shown in figure below, is that we can have variable length lists which are difficult to parse. We can create lists of any type, and the way these lists are encoded is that, inside the TLS specifications, we know how many bytes it takes to encode the total length of the lists. The network encoding of such a list is a struct with the length encoded as big-endian concatenated with the encoding of the elements of the lists.

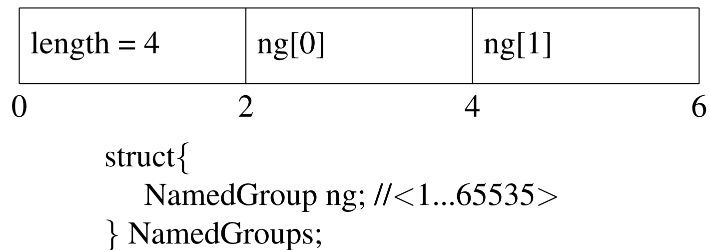


Figure 5.2: TLS parser variable length data structure.

To migrate TLS message parsers from OCaml to C, we have created a low-level parsing framework whose goal is to generate C parsers. Our approach is to generate TLS parser implementations in Low* from high level specification in F*. It is very important for this approach to keep separate the work we do under specification of how to generate parser implementations as generating efficient implementation is quite tricky and we have to rely on advanced features of F* such as *tactics*. However, its much easier to reason about the specification. We want to perform proofs that the parsers are correct according to the specifications and we only care about the fact that the implementation is correct because it is generated out of the pure specifications of the parser.

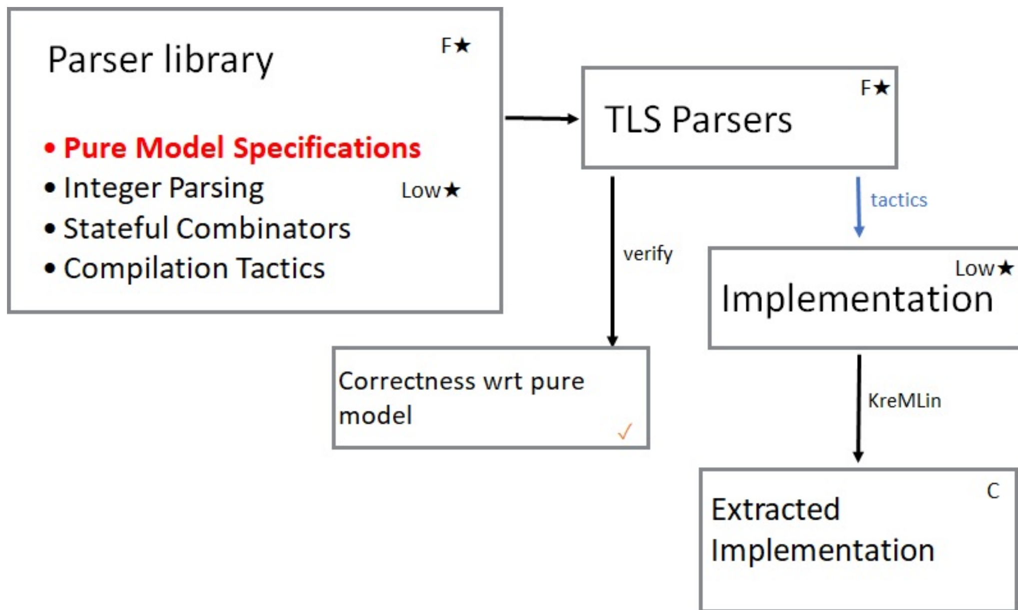


Figure 5.3: The TLS low-level parsing framework.

On top of this parsing framework, our work is actually about pure specifications of parsers and libraries to support them, in contrast to the above framework which is actually generating the state-full implementation of parsers and extracting them to C.

5.2 Correctness Specifications for Parsers

The correctness for parsers is defined as follows:

Property 3. Correctness is defined with respect to a pair of a serialiser and parser for type T.

$$\begin{aligned} \text{f:T} &\rightarrow \text{Tot bytes} \\ \text{g:bytes} &\rightarrow \text{Tot (option t)} \end{aligned}$$

where we define the correctness of parsing with respect to a serialiser of parser for a given high-level type T. A serialiser is a function that transforms T into bytes. It always expects an object of type T, therefore we cannot always create a serialised presentation.

For parsers, we want to interpret bytes as a concatenation of type T. This is not always possible and therefore it returns an optional T for types other than T. The reason for this is that not all bytes are ready for representation for elements of type T and the parser can fail on some malformed input.

Property 4. Serialiser Injectivity.

$$\forall \text{ a:T b:T. f a} = \text{f b} \rightarrow \text{a} = \text{b}$$

An important property for this function is injectivity. The two objects are the same if the two serialisations of two different operators of T are equal. This property is very important for security as we want to have unique representation for given objects of type T. It is also important because we actually sign the serialised representation of objects. As part of security proof, we can uniquely represent the given object of type T.

Property 5. Correctness: partial inverse.

$$\begin{aligned} \forall \text{ a:T. g (f a)} &= \text{Some a} \\ \text{Serialize} \mid> \text{Parse} &= \text{id} \end{aligned}$$

The correctness property that we care about for parsers is that if we try to parse the serialised elements, then we get exactly the same elements. Furthermore, if we serialise something that we parse successfully, then we get the same representation.

5.3 The New F⋆ Bytes Library

In order to implement these parser specifications, we have created a new bytes library for pure bytes F⋆. This library is meant to be used for the specifications of low-level implementation of parsers. The goals of this new library is to keep abstract the concrete representation of bytes. This is a significant improvement

compared with the previous version in which there was an exposed representation of bytes which was not actually the one used in the OCaml runtime of the program.

Due to the fact that the representation of bytes was exposed in the previous library, we could not rely on the properties of that particular representation if it was not proved for the actual implementation. It also contained runtime bugs which we strived to avoid at all costs. Another important point is that we wanted to unify the pure specifications that are used in TLS as compared to the one used in HACL*. Another main difference in the two libraries is that bytes were represented as machine integers in the previous version and were not compatible with the HACL* which use mathematical integers to represent them. In the newer version the two specifications are compatible which means that now we can also reason about serialising elements that come from HACL*. The second goal of the new bytes library is that we want to increase the automation and proof reliability. There is a lot of code inside the TLS code base that is dedicated to parser correctness proofs and is bogus, and we want to improve this code.

5.3.1 Parser

The type of parsers is

abstract type bytes = Seq.seq UInt8.t

In the newer version of F* bytes are defined internally as a sequence of abstract type of 8-bit machine integers. Earlier it was defined as strings which are practically very different than machine integers.

```
val append: b1:bytes -> b2:bytes ->
Tot (b:bytes{length b = length b1 + length b2})
let append (b1:bytes) (b2:bytes) = Seq.append b1 b2
val lemma_append_empty: b:bytes ->
Lemma (ensures (append b empty_bytes = b))
let rec lemma_append_empty (b:bytes) =
Seq.append_empty_r b
```

Listing 5.1: Bytes append function and lemmas.

We have also imported many existing lemmas from the previous library such as lemmas for concatenation and size. The above code contains a function for appending two sequences of bytes followed by a lemma and a function to append a sequence of bytes with an empty sequence. The *sub* function given below is used to crop a slice from the given sequence of bytes. It takes a sequence of bytes, locations of start and end bytes and returns a cropped sequence of bytes. It is followed by a lemma which proves that the length of the cropped sequence is either less than or equal to the length of the input sequence of bytes.

```
val sub: b:bytes -> s:nat{s < length b} ->
e:nat{e < length b / s <= e} ->
Tot (sub:bytes)
let sub b s e = Seq.slice b s e
let lemma_sub_length (b: bytes) (s:nat{s < length b})
(e:nat{e < length b / s <= e})
: Lemma (length (sub b s e) = e - s)
= lemma_len_slice b s e
```

Listing 5.2: Bytes subtract, index, and length functions and lemmas.

Two of the main new features of this library are that it has SMT patterns for the common lemmas and the support for machine integers. Earlier there was no support for machine integers which are required by HACL*, due to which numbers were encoded as strings instead of bytes, which is not the natural implementation of the encoding of numbers. Now, there is a difference between strings and bytes. There is also a support for unicode strings so that we could encode strings as bytes and the vice versa.

Next is an example of how to encode a variable length data structure. *vlbytes* is for encoding variable length bytes. If you are sure of the concatenated elements of the lists and want to add the length to it, then you can use the *vlbytes* function. Here it gives the size in bytes of the representation of the length of the variable length lists. and the concrete implementation is the functions for concatenation of the length of b with b where it is represented in big-endians.

```
val vlbytes: lsize:nat
-> b:B.bytes{B.repr_bytes (B.length b) <= lsize}
-> Tot (r:B.bytes{B.length r = lsize + B.length b})
let vlbytes lsize b = B.bytes_of_int lsize (B.length b) @| b
```

Listing 5.3: Transform and concatenate a natural number to bytes.

The *vlsplit* given next is a typical function that is used for parsing variable length fields inside a struct. When you have a field that is variable length, you read the length and split the remaining bytes in two. The first part is encoding of the variable length according to the length and the rest is the other field that you want to parse. You look at the *lsize* bytes and convert it to a number which corresponds to the length of the variable length field we are looking at. Then you check if it is less than the bytes you have to parse. If it is less then you split it according

to the length that you have interpreted. This can also fail if the encoding of the length is bigger than the bytes you have.

```
val vlsplit: lsize: nat{lsize <= 4}
-> vlb:B.bytes{lsize <= B.length vlb}
-> Tot (result (b:(B.bytes * B.bytes){
B.repr_bytes (B.length (fst b)) <= lsize /\
Seq.equal vlb (vlbytes lsize (fst b) @| (snd b))}))
♯set-options "–max_ifuel 2 –initial_ifuel 2"
let vlsplit lsize vlb =
let (vl,b) = B.split vlb lsize in
let l = B.int_of_bytes vl in
if l <= B.length b
then begin
let u, v = B.split b l in
B.append_assoc vl u v;
Correct (u,v)
end
else Error(AD_decode_error, perror __SOURCE_FILE__ __LINE__ "")
```

Listing 5.4: Parsing variable length fields.

Next is a concrete example of a simple datatype for simplified Diffie-Hellman Groups. Here the named groups are of three types, i.e. elliptical curves, finite fields, and an unknown type which is neither an elliptical curve nor a finite field. Because of the extensibility mechanism of the TLS, we cannot just avoid the fields which we do not understand. This is why when we define an unknown named group, we need to record the exact representation of that named group which is not in conflict with the representation, otherwise the injectivity proofs

would fail. Therefore, a declaration of type unknown in the named group should cover all the cases of elliptical curves and finite fields.

```
type ffdhe =
| FFDHE2048
| FFDHE3072
| FFDHE4096
| FFDHE6144
| FFDHE8192
type unknownNG =
u:(B.byte*B.byte){(let (b1,b2) = u in
(b1 = 0x00z ==> b2 <> 0x17z /\ b2 <> 0x18z /\ b2 <> 0x19z
/\ b2 <> 0x1dz /\ b2 <> 0x1ez) /\
(b1 = 0x01z ==> b2 <> 0x00z /\ b2 <> 0x01z /\ b2 <> 0x02z
/\ b2 <> 0x03z /\ b2 <> 0x04z))}
(** TLS 1.3 named groups for (EC)DHE key exchanges *)
type namedGroup =
| SEC of CoreCrypto.ec_curve
| FFDHE of ffdhe
| NG_UNKNOWN of unknownNG
```

Listing 5.5: Finite Field Diffie-Hellman group definitions.

5.4 Summary

We have implemented a new bytes library for F*. We also have ported the most significant message processing modules of miTLS such as *Parse* and *TLSConstants* to the new bytes library. We have also modified the corresponding proofs

for these modules using the new bytes library. We are in a process of porting the rest of the miTLS and removing the dependency on the old bytes model in the *Platform Library*. As mentioned earlier, we want to use the parser specifications as an input to the parsing framework shown in Figure 5.3 in order to generate efficient code for the parsers.

Part III

Appendixes

Appendix A

Bytes Library for TLS Message Parsers

This appendix contains the code for the new *Bytes library* for F* and Project Everest which is disccused in chapters 4 and 5. It consists of 450 lines approximately and replaces the previous unsound *Platform library*.

```
1   module Parse
2
3   open FStar.HyperStack.All
4
5   open Platform.Error
6   open TLSError
7
8   module HH = FStar.HyperHeap
9   module HS = FStar.HyperStack
10  module ST = FStar.HyperStack.ST
11  module B = FStar.Bytes
12
13  let op_At_Bar (b1:B.bytes) (b2:B.bytes) = B.append b1 b2
14
15  (** This file should be split in 3 different modules:
16      - Regions: for global table regions
17      - Format: for generic formatting functions
18      - DHFormat: for (EC)DHE-specific formatting
19  *)
20
21
22  (** Begin Module Regions *)
23
24  //type fresh_subregion r0 r h0 h1 =
    ST.stronger_fresh_region r h0 h1 /\ ST.extends r r0
25
26  (** Regions and colors for objects in memory *)
27  let tls_color = -1
28  let epoch_color = 1
29  let hs_color = 2
30
31  let is_tls_rgn r   = HH.color r = tls_color
32  let is_epoch_rgn r = HH.color r = epoch_color
33  let is_hs_rgn r    = HH.color r = hs_color
34
35  (*
36   * AR: Adding the eternal region predicate.
37   * Strengthening the predicate because at some places, the
     code uses HH.parent.
38   *)
39  let rgn       = r:HH.rid{r<>HH.root
```

```
40                              /\ (forall (s:HH.rid).{:pattern
                                HS.is_eternal_region s}
                                HS.is_above s r ==>
                                HS.is_eternal_region s)}
41   let tls_rgn   = r:rgn{is_tls_rgn r}
42   let epoch_rgn = r:rgn{is_epoch_rgn r}
43   let hs_rgn    = r:rgn{is_hs_rgn r}
44
45   let tls_region : tls_rgn = new_colored_region HH.root
     tls_color
46
47   let tls_tables_region : (r:tls_rgn{HH.parent r =
     tls_region}) =
48       new_region tls_region
49
50
51   (** End Module Regions *)
52
53
54   (** Begin Module Format *)
55
56   // basic parsing and formatting---an attempt at splitting
     TLSConstant.
57
58   type pinverse_t (#a:Type) (#b:Type) ($f:(a -> Tot b)) = b
     -> Tot (result a)
59
60   unfold type lemma_inverse_g_f (#a:Type) (#b:Type) ($f:a ->
     Tot b) ($g:b -> Tot (result a)) (x:a) =
61     g (f x) == Correct x
62
63   unfold type lemma_pinverse_f_g (#a:Type) (#b:Type) (r:b ->
     b -> Type) ($f:a -> Tot b) ($g:b -> Tot (result a)) (y:b)
     =
64     Correct? (g y) ==> r (f (Correct?._0 (g y))) y
65
66
67   (** Transforms a sequence of natural numbers into bytes *)
68   val bytes_of_seq: n:nat{B.repr_bytes n <= 8 } -> Tot
     (b:B.bytes{B.length b <= 8})
69   let bytes_of_seq sn = B.bytes_of_int 8 sn
70
71   (** Transforms bytes into a sequence of natural numbers *)
```

```
72  val seq_of_bytes: b:B.bytes{ B.length b <= 8 } -> Tot nat
73  let seq_of_bytes b = B.int_of_bytes b
74
75  (** Transform and concatenate a natural number to bytes *)
76  val vlbytes: lSize:nat -> b:B.bytes{B.repr_bytes (B.length
    b) <= lSize} -> Tot (r:B.bytes{B.length r = lSize +
    B.length b})
77  let vlbytes lSize b = B.bytes_of_int lSize (B.length b) @|
    b
78
79  // avoiding explicit applications of the representation
    lemmas
80  let vlbytes1 (b:B.bytes {B.length b < pow2 8}) =
    B.lemma_repr_bytes_values (B.length b); vlbytes 1 b
81  let vlbytes2 (b:B.bytes {B.length b < pow2 16}) =
    B.lemma_repr_bytes_values (B.length b); vlbytes 2 b
82
83  val vlbytes_trunc: lSize:nat -> b:B.bytes ->
84    extra:nat{B.repr_bytes (B.length b + extra) <= lSize} ->
85    r:B.bytes{B.length r == lSize + B.length b}
86  let vlbytes_trunc lSize b extra =
87    B.bytes_of_int lSize (B.length b + extra) @| b
88
89  let vlbytes_trunc_injective
90    (lSize: nat)
91    (b1: B.bytes)
92    (extra1: nat { B.repr_bytes (B.length b1 + extra1) <=
      lSize } )
93    (s1: B.bytes)
94    (b2: B.bytes)
95    (extra2: nat { B.repr_bytes (B.length b2 + extra2) <=
      lSize } )
96    (s2: B.bytes)
97  : Lemma
98    (requires ((vlbytes_trunc lSize b1 extra1 @| s1) =
      (vlbytes_trunc lSize b2 extra2 @| s2)))
99    (ensures (B.length b1 + extra1 == B.length b2 + extra2
      /\ b1 @| s1 == b2 @| s2))
100 = let l1 = B.bytes_of_int lSize (B.length b1 + extra1) in
101   let l2 = B.bytes_of_int lSize (B.length b2 + extra2) in
102   B.append_assoc l1 b1 s1;
103   B.append_assoc l2 b2 s2;
104   B.lemma_append_inj l1 (b1 @| s1) l2 (b2 @| s2);
```

```
105   B.int_of_bytes_of_int lSize (B.length b1 + extra1);
106   B.int_of_bytes_of_int lSize (B.length b2 + extra2)
107
108 (** Lemmas associated to bytes manipulations *)
109 val lemma_vlbytes_len : i:nat -> b:B.bytes{B.repr_bytes
    (B.length b) <= i}
110   -> Lemma (ensures (B.length (vlbytes i b) = i + B.length
      b))
111 let lemma_vlbytes_len i b = ()
112
113
114 val lemma_vlbytes_inj_strong : i:nat
115   -> b:B.bytes{B.repr_bytes (B.length b) <= i}
116   -> s:B.bytes
117   -> b':B.bytes{B.repr_bytes (B.length b') <= i}
118   -> s':B.bytes
119   -> Lemma (requires ((vlbytes i b @| s) = (vlbytes i b'
      @| s')))
120           (ensures (b == b' /\ s == s'))
121
122 let lemma_vlbytes_inj_strong i b s b' s' =
123   let l = B.bytes_of_int i (B.length b) in
124   let l' = B.bytes_of_int i (B.length b') in
125   B.append_assoc l b s;
126   B.append_assoc l' b' s';
127   B.lemma_append_inj l (b @| s) l' (b' @| s');
128   B.int_of_bytes_of_int i (B.length b);
129   B.int_of_bytes_of_int i (B.length b');
130   B.lemma_append_inj b s b' s'
131
132 val lemma_vlbytes_inj : i:nat
133   -> b:B.bytes{B.repr_bytes (B.length b) <= i}
134   -> b':B.bytes{B.repr_bytes (B.length b') <= i}
135   -> Lemma (requires ((vlbytes i b) = (vlbytes i b')))
136           (ensures (b == b'))
137
138 let lemma_vlbytes_inj i b b' =
139   lemma_vlbytes_inj_strong i b B.empty_bytes b'
      B.empty_bytes
140
141 val vlbytes_length_lemma: n:nat -> a:B.bytes{B.repr_bytes
    (B.length a) <= n} -> b:B.bytes{B.repr_bytes (B.length b)
    <= n} ->
```

```
142   Lemma (requires ((B.slice (vlbytes n a) 0 n) = (B.slice
      (vlbytes n b) 0 n)))
143         (ensures (B.length a = B.length b))
144
145 let vlbytes_length_lemma n a b = admit()
146
147 #set-options "--max_ifuel 1 --initial_ifuel 1 --max_fuel 0
    --initial_fuel 0"   //need to reason about length
148
149 val vlsplit: lSize:nat{lSize <= 4}
150   -> vlb:B.bytes{lSize <= B.length vlb}
151   -> Tot (result (b:(B.bytes * B.bytes){
152         B.repr_bytes (B.length (fst b)) <= lSize /\
153         Seq.equal vlb (vlbytes lSize (fst b) @| (snd
            b))}))
154
155 #set-options "--max_ifuel 2 --initial_ifuel 2"
156 let vlsplit lSize vlb =
157   let (vl,b) = B.split vlb lSize in
158   let l = B.int_of_bytes vl in
159   if l <= B.length b
160   then begin
161     let u, v = B.split b l in
162     B.append_assoc vl u v;
163     Correct (u,v)
164    end
165   else Error(AD_decode_error, perror __SOURCE_FILE__
      __LINE__ "")
166
167 val vlparse: lSize:nat{lSize <= 4} -> vlb:B.bytes{lSize <=
    B.length vlb}
168   -> Pure (r:result B.bytes)
169   (requires (True))
170   (ensures fun r -> Correct? r ==>
171     (let b = Correct?._0 r in B.repr_bytes (B.length b) <=
        lSize /\ vlb = (vlbytes lSize b)))
172
173 let vlparse lSize vlb =
174   let vl,b = B.split vlb lSize in
175   if B.int_of_bytes vl = B.length b then Correct b
176   else Error(AD_decode_error, perror __SOURCE_FILE__
      __LINE__ "")
177
```

```
178 val vlparse_vlbytes: lSize:nat{lSize <= 4} ->
    vlb:B.bytes{B.repr_bytes (B.length vlb) <= lSize} -> Lemma
179   (requires (True))
180   (ensures (vlparse lSize (vlbytes lSize vlb) == Correct
      vlb))
181   [SMTPat (vlparse lSize (vlbytes lSize vlb))]
182 
183 #reset-options "--initial_ifuel 2 --initial_fuel 2"
184 let vlparse_vlbytes lSize vlb =
185   let b' = vlbytes lSize vlb in
186   assert(b' = B.bytes_of_int lSize (B.length vlb) @| vlb);
187   let vl, b = B.split b' lSize in
188   B.lemma_append_inj vl b (B.bytes_of_int lSize (B.length
      vlb)) vlb;
189   assert (vl = (B.bytes_of_int lSize (B.length vlb)));
190   B.int_of_bytes_of_int lSize (B.length vlb);
191   let x = vlparse lSize b' in
192   let b'' = Correct?._0 x in
193   assert(x == vlparse lSize (vlbytes lSize vlb))
194 
195 val uint16_of_bytes:
196   b:B.bytes{B.length b == 2} ->
197   n:UInt16.t{B.repr_bytes (UInt16.v n) <= 2 /\
      B.bytes_of_int 2 (UInt16.v n) == b}
198 
199 let uint16_of_bytes b =
200   let n = B.int_of_bytes b in
201   assert_norm (pow2 16 == 65536);
202   B.lemma_repr_bytes_values n;
203   B.int_of_bytes_of_int 2 n;
204   let r = UInt16.uint_to_t n in
205   assert(UInt16.v r = n);
206   assert(B.repr_bytes (UInt16.v r) <= 2);
207   r
208 
209 val uint32_of_bytes:
210   b:B.bytes{B.length b == 4} ->
211   n:UInt32.t{B.repr_bytes (UInt32.v n) <= 4 /\
      B.bytes_of_int 4 (UInt32.v n) == b}
212 
213 let uint32_of_bytes b =
214   let n = B.int_of_bytes b in
215   assert_norm (pow2 32 == 4294967296);
```

```
216    B.lemma_repr_bytes_values n;
217    UInt32.uint_to_t n
218
219  let bytes_of_uint32 (n:UInt32.t) : Tot (B.lbytes 4) =
220    let n = UInt32.v n in
221    B.lemma_repr_bytes_values n;
222    B.bytes_of_int 4 n
223
224  let bytes_of_uint16 (n:UInt16.t) : Tot (B.lbytes 2) =
225    let n = UInt16.v n in
226    B.lemma_repr_bytes_values n;
227    B.bytes_of_int 2 n
228
229  (** End Module Format *)
230
231
232  (** Begin Module DHFormat *)
233
234  // floating crypto definitions
235
236  (** Finite Field Diffie-Hellman group definitions *)
237  type ffdhe =
238    | FFDHE2048
239    | FFDHE3072
240    | FFDHE4096
241    | FFDHE6144
242    | FFDHE8192
243
244  type unknownNG =
245    u:(B.byte*B.byte){(let (b1,b2) = u in
246      (b1 = 0x00z ==> b2 <> 0x17z /\ b2 <> 0x18z /\ b2 <>
       0x19z
247                   /\ b2 <> 0x1dz /\ b2 <> 0x1ez) /\
248      (b1 = 0x01z ==> b2 <> 0x00z /\ b2 <> 0x01z /\ b2 <>
       0x02z
249                   /\ b2 <> 0x03z /\ b2 <> 0x04z))}
250
251  (** TLS 1.3 named groups for (EC)DHE key exchanges *)
252  type namedGroup =
253    | SEC of CoreCrypto.ec_curve
254    | FFDHE of ffdhe
255    | NG_UNKNOWN of unknownNG
256
```

```
257 (*
258  * We only seem to be using these two named groups
259  * irrespective of whether it's TLS 12 or 13
260  *)
261 type valid_namedGroup = x:namedGroup{SEC? x \/ FFDHE? x}
262
263 (** Serializing function for (EC)DHE named groups *)
264 val namedGroupBytes: namedGroup -> Tot (B.lbytes 2)
265 let namedGroupBytes ng =
266   let open CoreCrypto in
267   match ng with
268   | SEC ec ->
269     begin
270     match ec with
271     | ECC_P256    -> B.abyte2 (0x00z, 0x17z)
272     | ECC_P384    -> B.abyte2 (0x00z, 0x18z)
273     | ECC_P521    -> B.abyte2 (0x00z, 0x19z)
274     | ECC_X25519  -> B.abyte2 (0x00z, 0x1dz)
275     | ECC_X448    -> B.abyte2 (0x00z, 0x1ez)
276     end
277   | FFDHE dhe ->
278     begin
279     match dhe with
280     | FFDHE2048   -> B.abyte2 (0x01z, 0x00z)
281     | FFDHE3072   -> B.abyte2 (0x01z, 0x01z)
282     | FFDHE4096   -> B.abyte2 (0x01z, 0x02z)
283     | FFDHE6144   -> B.abyte2 (0x01z, 0x03z)
284     | FFDHE8192   -> B.abyte2 (0x01z, 0x04z)
285     end
286   | NG_UNKNOWN u  -> B.abyte2 u
287
288 (* TODO: move to Platform.Bytes *)
289 let abyte2_inj x1 x2 : Lemma
290   (B.abyte2 x1 == B.abyte2 x2 ==> x1 == x2)
291   [SMTPat (B.abyte2 x1); SMTPat (B.abyte2 x2)]
292 = let s1 = B.abyte2 x1 in
293   let s2 = B.abyte2 x2 in
294   assert (x1 == (B.index s1 0, B.index s1 1));
295   assert (x2 == (B.index s2 0, B.index s2 1))
296
297 val namedGroupBytes_is_injective: ng1:namedGroup ->
    ng2:namedGroup ->
```

```
298    Lemma (requires ((namedGroupBytes ng1) =
       (namedGroupBytes ng2)))
299          (ensures (ng1 == ng2))
300  let namedGroupBytes_is_injective ng1 ng2 =
301    admit()
302
303  (** Parsing function for (EC)DHE named groups *)
304  val parseNamedGroup: pinverse_t namedGroupBytes
305  let parseNamedGroup b =
306    let open CoreCrypto in
307    match B.cbyte2 b with
308    | (0x00z, 0x17z) -> Correct (SEC ECC_P256)
309    | (0x00z, 0x18z) -> Correct (SEC ECC_P384)
310    | (0x00z, 0x19z) -> Correct (SEC ECC_P521)
311    | (0x00z, 0x1dz) -> Correct (SEC ECC_X25519)
312    | (0x00z, 0x1ez) -> Correct (SEC ECC_X448)
313    | (0x01z, 0x00z) -> Correct (FFDHE FFDHE2048)
314    | (0x01z, 0x01z) -> Correct (FFDHE FFDHE3072)
315    | (0x01z, 0x02z) -> Correct (FFDHE FFDHE4096)
316    | (0x01z, 0x03z) -> Correct (FFDHE FFDHE6144)
317    | (0x01z, 0x04z) -> Correct (FFDHE FFDHE8192)
318    | u -> Correct (NG_UNKNOWN u)
319
320  (** Lemmas for named groups parsing/serializing inversions
     *)
321  #set-options "--max_ifuel 10 --max_fuel 10"
322  val inverse_namedGroup: x:_ -> Lemma
323    (requires True)
324    (ensures lemma_inverse_g_f namedGroupBytes
       parseNamedGroup x)
325    [SMTPat (parseNamedGroup (namedGroupBytes x))]
326  let inverse_namedGroup x = ()
327
328  val pinverse_namedGroup: x:_ -> Lemma
329    (requires True)
330    (ensures (lemma_pinverse_f_g Seq.equal namedGroupBytes
       parseNamedGroup x))
331    [SMTPat (namedGroupBytes (Correct?._0 (parseNamedGroup
       x)))]
332  let pinverse_namedGroup x = ()
333
334  #set-options "--initial_fuel 2 --initial_ifuel 2"
```

```
335 private let lemma_namedGroupBytes_injective
    (ng1:namedGroup) (ng2:namedGroup)
336   : Lemma (requires (namedGroupBytes ng1 = namedGroupBytes
      ng2))
337           (ensures (ng1 = ng2))
338   =
339   let open CoreCrypto in
340   let nb1 = namedGroupBytes ng1 in
341   let nb2 = namedGroupBytes ng2 in
342   match ng1, ng2 with
343   | SEC ec1, SEC ec2 ->
344     if ec1 = ec2 then ()
345     else assert(B.index nb1 0 = 0x00z /\ B.index nb2 0 =
        0x00z)
346   | FFDHE dhe1, FFDHE dhe2 ->
347     if dhe1 = dhe2 then ()
348     else assert(B.index nb1 0 = 0x01z /\ B.index nb2 0 =
        0x01z)
349   | NG_UNKNOWN u1, NG_UNKNOWN u2 ->
350     abyte2_inj u1 u2
351   | _ -> assert(B.index nb1 0 = B.index nb2 0 /\ B.index
      nb1 1 = B.index nb2 1)
352 #reset-options
353
354 #set-options "--max_ifuel 2 --max_fuel 2"
355 private val namedGroupsBytes0: groups:list namedGroup
356   -> Tot (b:B.bytes { B.length b == op_Multiply 2
      (List.Tot.length groups)})
357 let rec namedGroupsBytes0 groups =
358   match groups with
359   | [] -> B.empty_bytes
360   | g::gs ->
361     B.lemma_len_append (namedGroupBytes g)
        (namedGroupsBytes0 gs);
362     namedGroupBytes g @| namedGroupsBytes0 gs
363 #reset-options
364
365 #set-options "--initial_fuel 2 --initial_ifuel 2"
366 private
367 let rec namedGroupsBytes0_is_injective
368   (groups1 groups2: list namedGroup)
369 : Lemma
```

```
370   (requires ((namedGroupsBytes0 groups1) =
      (namedGroupsBytes0 groups2)))
371   (ensures (groups1 == groups2))
372 = match groups1, groups2 with
373   | [], [] -> ()
374   | g1::groups1', g2::groups2' ->
375     assert((namedGroupBytes g1) @| (namedGroupsBytes0
        groups1') = (namedGroupBytes g2) @| (namedGroupsBytes0
        groups2'));
376     B.lemma_append_inj (namedGroupBytes g1)
        (namedGroupsBytes0 groups1') (namedGroupBytes g2)
        (namedGroupsBytes0 groups2');
377     lemma_namedGroupBytes_injective g1 g2;
378     namedGroupsBytes0_is_injective groups1' groups2'
379
380 (** Serialization function for a list of named groups *)
381 val namedGroupsBytes: groups:list
    namedGroup{List.Tot.length groups < 65536/2}
382   -> Tot (b:B.bytes { B.length b = 2 + op_Multiply 2
      (List.Tot.length groups)})
383
384 let namedGroupsBytes groups =
385   let gs = namedGroupsBytes0 groups in
386   B.lemma_repr_bytes_values (B.length gs);
387   vlbytes 2 gs
388
389 let namedGroupsBytes_is_injective
390   (groups1: list namedGroup { List.Tot.length groups1 <
      65536/2 } )
391   (s1: B.bytes)
392   (groups2: list namedGroup { List.Tot.length groups2 <
      65536/2 } )
393   (s2: B.bytes)
394 : Lemma
395   (requires (Seq.equal (namedGroupsBytes groups1 @| s1)
      (namedGroupsBytes groups2 @| s2)))
396   (ensures (groups1 == groups2 /\ s1 == s2))
397 = let gs1 = namedGroupsBytes0 groups1 in
398   B.lemma_repr_bytes_values (B.length gs1);
399   let gs2 = namedGroupsBytes0 groups2 in
400   B.lemma_repr_bytes_values (B.length gs2);
401   lemma_vlbytes_inj_strong 2 gs1 s1 gs2 s2;
402   namedGroupsBytes0_is_injective groups1 groups2
```

```
403
404 private val parseNamedGroups0: b:B.bytes -> l:list
    namedGroup
405   -> Tot (result (groups:list namedGroup{List.Tot.length
      groups = List.Tot.length l + B.length b / 2}))
406   (decreases (B.length b))
407 let rec parseNamedGroups0 b groups =
408   if B.length b > 0 then
409     if B.length b >= 2 then
410       let (ng, bytes) = B.split b 2 in
411       B.lemma_split b 2;
412       match parseNamedGroup ng with
413       |Correct ng ->
414         let groups' = ng :: groups in
415         parseNamedGroups0 bytes groups'
416       | Error z    -> Error z
417     else Error (AD_decode_error, perror __SOURCE_FILE__
        __LINE__ "")
418   else
419    let grev = List.Tot.rev groups in
420    assume (List.Tot.length grev == List.Tot.length
       groups);
421    Correct grev
422
423 (** Parsing function for a list of named groups *)
424 val parseNamedGroups: b:B.bytes { 2 <= B.length b /\
    B.length b < 65538 }
425   -> Tot (result (groups:list namedGroup{List.Tot.length
      groups = (B.length b - 2) / 2}))
426
427 let parseNamedGroups b =
428   match vlparse 2 b with
429   | Correct b' -> parseNamedGroups0 b' []
430   | _ -> Error(AD_decode_error, perror __SOURCE_FILE__
      __LINE__ "Failed to parse named groups")
431
432 (* End Module DHFormat *)
433
```

Bibliography

[1] https://regex101.com.

[2] The open group base specification issue 6 ieee std 1003.1 2004 edition. http://pubs.opengroup.org/olinepubs/009695399/basedefs/xbd_chap09.html, 2004.

[3] A. V. Aho. Algorithms for Finding Patterns in Strings. In J. van Leeuwen, editor, *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 255–300. Elsevier, 1990.

[4] J. B. Almeida, N. Moriera, D. Pereira, and S. M. de Sousa. Partial Derivative Automata Formalized in Coq. In *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA)*, volume 6482 of *LNCS*, pages 59–68, 2010.

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

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

[7] M. Becchi and P. Crowley. Extending Finite Automata to Efficiently Match Perl-compatible Regular Expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*, pages 25:1–25:12. ACM, 2008.

[8] M. Berglund, W. Bester, and B. van der Merwe. Formalising Boost POSIX Regular Expression Matching. Accepted for publication at ICTAC'18.

[9] M. Berglund, F. Drewes, and B. van der Merwe. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. In *Proc. of the 14th International Conference on Automata and Formal Languages*, pages 109–123. Springer Verlag, 2014.

[10] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 535–552. IEEE, 2015.

[11] M. Bezem, J.W. Klop, and R. de Vrijer. *Term Rewriting Systems*. Cambridge Tracts in Theoretica. Cambridge University Press, 2003.

[12] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub. Implementing TLS with Verified Cryptographic Security. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459. IEEE, 2013.

[13] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Y. Strub, and S. Zanella-Béguelin. Proving the TLS Handshake Secure (as it is). In *International Cryptology Conference*, pages 235–255. Springer, 2014.

[14] W. Bokslag. The Problem of Popular Primes: Logjam. *arXiv preprint arXiv:1602.02396*, 2016.

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

[16] C. Campeanu, K. Salomaa, and S. Yu. A Formal Study of Practical Regular Expressions. *International Journal of Foundations of Computer Science*, 14:1007–1018, 2003.

[17] G. Carlucci, L. De Cicco, and S. Mascolo. HTTP over UDP: an Experimental Investigation of QUIC. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 609–614. ACM, 2015.

[18] P. Caron, J.-M. Champarnaud, and L. Mignot. A General Framework for the Derivation of Regular Expressions. *RAIRO - Theoretical Informatics and Applications*, 48(3):281–305, 2014.

[19] H. Chen and S. Yu. Derivatives of Regular Expressions and an Application. In *Proc. in the International Workshop on Theoretical Computer Science (WTCS)*, volume 7160 of *LNCS*, pages 343–356, 2012.

[20] T. Coquand and V. Siles. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Proc. of the 1st Conference on Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 119–134, 2011.

[21] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, N. Swamy, and S. Zanella-Beguelin. Towards a Provably Secure Implementation of TLS 1.3. July 2016.

[22] C. Doczkal, J.-O. Kaiser, and G. Smolka. A Constructive Theory of Regular Languages in Coq. In *Proc. of the International Conference on Certified Programs and Proofs*, volume 8307, pages 82–97. Springer, 2013.

[23] A. Frisch and L. Cardelli. Greedy Regular Expression Matching. In *Proc. of the 31st International Conference on Automata, Languages and Programming (ICALP)*, volume 3142 of *LNCS*, pages 618–629, 2004.

[24] J. Goyvaerts. *Regular Expressions Cookbook: Detailed Solutions in Eight Programming Languages*. O'Reilly Media, 2012.

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

[26] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

[27] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, 2005.

[28] IETF.org. The transport layer security protocol version 1.2.

[29] J. Kirrage, A. Rathnayake, and H. Thielecke. Static Analysis for Regular Expression Denial-of-Service Attacks. In *In Proc. of the International Conference on Network and System Security*, pages 135–148. Springer Verlag, 2013.

[30] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Annals of Mathematics Studies*, 34:3–41, 1951.

[31] D. Kozen. *Automata and Computability*. Springer Verlag, 1997.

[32] A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *Journal of Automated Reasoning*, 49:95–106, 2012.

[33] N. R. Krishnaswami and J. Yallop. A Typed, Algebraic Approach to Parsing. unpublished.

[34] C. Kuklewicz. Regex Posix. https://wiki.haskell.org/Regex_Posix.

[35] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 183–196. ACM, 2017.

[36] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller. Finding Error Handling Bugs in OpenSSL using Coccinelle. In *Dependable Computing Conference (EDCC), 2010 European*, pages 191–196. IEEE, 2010.

[37] P. Libič, L. Bulej, V. Horký, and P. Tůma. Estimating the Impact of Code Additions on Garbage Collection Overhead. In M. Beltrán, W. Knottenbelt, and J. Bradley, editors, *Computer Performance Engineering*, pages 130–145, Cham, 2015. Springer International Publishing.

[38] F. McNaughton and H. Yamada. Regular expressions and state graphs for automata. EC-9:39–47, 1960.

[39] P. Morrissey, N. P. Smart, and B. Warinschi. A Modular Security Analysis of the TLS Handshake Protocol. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 55–73. Springer, 2008.

[40] N. Murugesan and O. V. Shanmuga Sundaram. Some Properties of Brzozowski Derivatives of Regular Expressions. *International Journal of Computer Trends and Technology*, 13(1):29–33, 2014.

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

[42] S. Okui and T. Suzuki. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. Technical report, University of Aizu, 2013.

[43] S. Owens, J. H. Reppy, and A. Turon. Regular-Expression Derivatives Re-Examined. *Journal of Functional Programming*, 19(2):173–190, 2009.

[44] S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.

[45] L. C. Paulson. A Formalisation of Finite Automata Using Hereditarily Finite Sets. In *Proc. of the 25th International Conference on Automated Deduction (CADE)*, volume 9195 of *LNAI*, pages 231–245, 2015.

[46] V. Paxson. Bro. `https://www.bro.org`, 1994–2018.

[47] T. Reps. "Maximal-Munch" Tokenization in Linear Time. *ACM Transaction of Programming Language Systems*, 20(2):259–273, 1998.

[48] R. Ribeiro and A. Du Bois. Certified Bit-Coded Regular Expression Parsing. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, SBLP 2017, pages 4:1–4:8. ACM, 2017.

[49] M. Roesch. Snort. `https://www.snort.org`, 1998–2018.

[50] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[51] J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008.

[52] P. Sirohi, A. Agarwal, and S. Tyagi. A Comprehensive Study on Security Attacks on the SSL/TLS Protocol. In *Next Generation Computing Technologies (NGCT), 2016 2nd International Conference on*, pages 893–898. IEEE, 2016.

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

[54] M. Sulzmann and P.V. Steenhoven. A Flexible and Efficient ML Lexer Tool Based on Extended Regular Expression Submatching. *Compiler Construction*, pages 174–191, 2014.

[55] 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)*, volume 8977 of *LNCS*, pages 275–286, 2015.

[56] K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[57] D. Traytel and T. Nipkow. A Verified Decision Procedure for MSO on Words Based on Derivatives of Regular Expressions. In *Proc. of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 3–12, 2013.

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

[59] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *In Proc. of the International Conference on Implementation and Application of Automata*, pages 322–334. Springer Verlag, 2016.

[60] C. Wu, X. Zhang, and C. Urban. A Formalisation of the Myhill-Nerode Theorem based on Regular Expressions. *Journal of Automatic Reasoning*, 52(4):451–480, 2014.