Semantics, Analysis and Security of Backtracking Regular Expression Matchers

By

Asiri Rathnayake

A thesis submitted to the
University of Birmingham
for the degree of
Doctor of Philosophy

Computer Science
University of Birmingham
August 2014

Abstract

Regular expressions are ubiquitous in computer science. Originally defined by Kleene in 1956, they have become a staple of the computer science undergraduate curriculum. Practical applications of regular expressions are numerous, ranging from compiler construction through smart text editors to network intrusion detection systems. Despite having been vigorously studied and formalized in many ways, recent practical implementations of regular expressions have drawn criticism for their use of a non-standard backtracking algorithm. In this research, we investigate the reasons for this deviation and develop a semantics view of regular expressions that formalizes the backtracking paradigm. In the process we discover a novel static analysis capable of detecting exponential runtime vulnerabilities; an extremely undesired reality of backtracking regular expression matchers.

Acknowledgments

The work presented in this thesis was conducted jointly with my supervisor, Dr. Hayo Thielecke. I would also like to acknowledge the constructive discussions with Dr. Tom Chothia, Dr. Marco Cova, James Kirrage and Bram Geron, which yielded valuable insights into various aspects of the research. I am extremely grateful for the School of Computer Science, University of Birmingham for funding this research.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Regular expressions form a minimalistic language of pattern-matching constructs. Originally defined in Kleene's work on the foundations of computation, they have become ubiquitous in computing. The study of regular expressions has been a vital chapter of computer science undergraduate curricula for many years. They have also been explored by researchers in many other directions, such as efficient automata construction/representation, language derivatives, parse extraction and type systems. This ubiquitous nature of regular expressions might make one believe that they represent a solved problem, a success story of computer science or a thing of the past. Such prejudice is not entirely unwarranted given that traditional automata theory is well understood, time-tested over decades in relation to compiler construction. Given this reputation, it is quite important to identify the goals of another take on the subject.

1.1 Background

Current understanding of regular expressions is rooted in automata theory, as taught in most undergraduate courses. In order to appreciate the approach taken in the present work, we need to recap some of the most relevant topics from automata theory. We will only remind ourselves of these concepts, for a more thorough treatment, one may refer to the literature review in the next chapter or the well-known text books on the subject [HU79, Sip96].

1.1.1 Regular languages

Regular expressions represent regular languages, the least expressive class of languages from the Chomsky Hierarchy [Cho56].

Definition 1.1.1 The regular language denoted by a regular expression e is defined as follows:

$$\begin{aligned}
\mathcal{L}(\phi) &= \emptyset \\
\mathcal{L}(\varepsilon) &= \{\varepsilon\} \\
\mathcal{L}(a) &= \{a\} \quad a \in \Sigma \\
\mathcal{L}(e_1 e_2) &= \{xy | x \in \mathcal{L}(e_1) \wedge y \in \mathcal{L}(e_2)\} \\
\mathcal{L}(e_1 \mid e_2) &= \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \\
\mathcal{L}(e^*) &= \{\varepsilon\} \cup \mathcal{L}(ee^*)
\end{aligned}$$

Essentially, each regular expression defines a (potentially infinite) set of strings which constitute the corresponding regular language. Note that ε is a regular expression constant which corresponds to the language of the

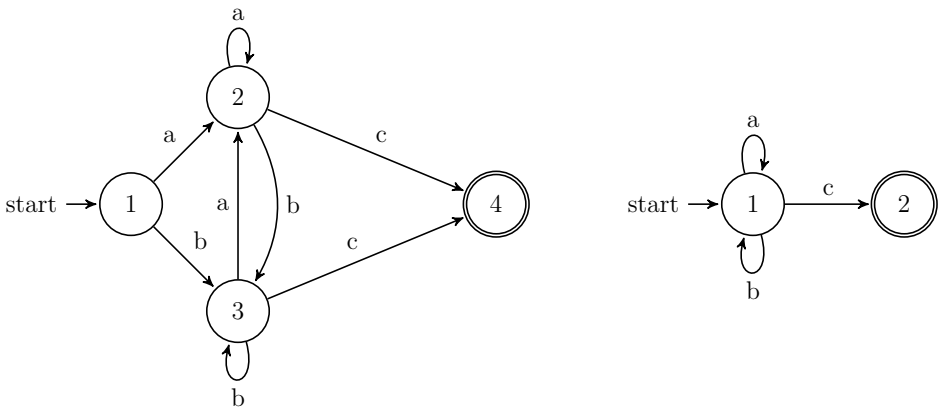
empty string $\{\varepsilon\}$.

1.1.2 Automata

Any regular expression can be represented by an automaton, a computational device capable of recognising the set of strings belonging to a regular language. The relationship between a regular expression e, an equivalent automaton N_e and the corresponding regular language $\mathcal{L}(e)$ may be specified as follows:

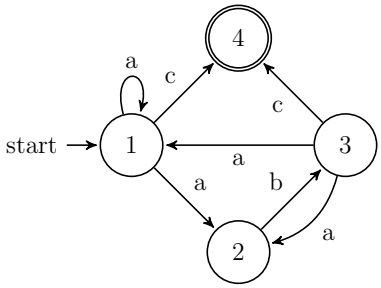
$$\mathcal{L}(e) = \{w | w \text{ is accepted by } N_e\}$$

Deterministic Finite Automata (DFA) are state machines with deterministic state transitions on input symbols. Following diagram illustrates two example DFAs corresponding to the regular expression $(a \mid b)^*c$:

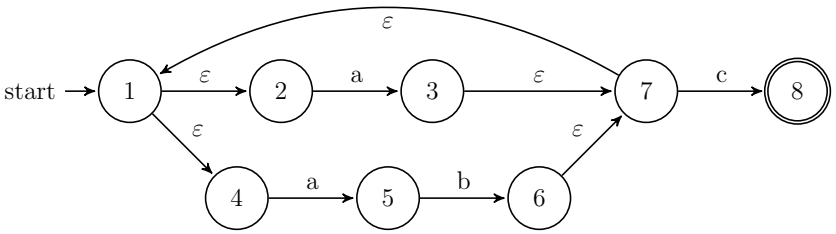


The DFA to the right is the minimal-state DFA corresponding to the said expression.

Non-deterministic Finite Automata (NFA) on the other hand contain non-deterministic transitions. The diagram below shows an NFA corresponding to the regular expression $(a \mid ab)^*c$:



Here the states 1 and 3 have nondeteministic transitions on the input symbol a. A variant of NFA known as ε-NFA also accommodates silent transitions; state transitions that take place without consuming input. Following diagram illustrates an ε-NFA corresponding to the same expression:



Importantly, all forms of automata have equal expressive power; they can represent any regular language (and regular languages only). There are standard algorithms for constructing different kinds of automata and transforming from one kind to the other. We defer to theory texts [HU79, Sip96, ALSU07] for details on automata construction / transformation algorithms, resolving to discuss some of those algorithms only when they become relevant to the main research.

1.1.3 Fundamental matching algorithms

The problem of matching is to validate if a given string belongs to a regular language represented by some automaton. The general idea is to simulate

the automaton against the input string while keeping track of the current state. For DFA the algorithm is straightforward, we start with the initial state and keep calculating the next active state by following the transition corresponding to the next input symbol. If an accepting state is reached at the end of the input string, the string is accepted, otherwise it is rejected. This algorithm has linear complexity in both time and space.

For NFA the algorithm needs to keep track of a set of active states. As each input symbol is processed, a new set of active states is computed by exploring all the outgoing transitions of the current set of active states (including ε transitions). This is known as the lockstep algorithm [Tho68]; an on-the-fly subset construction [ALSU07], where each set of active states corresponds to some DFA state. This algorithm has $\mathcal{O}(mn)$ time complexity and $\mathcal{O}(m)$ space complexity, where n is the length of the input string and m is the size of the NFA. It is possible to optimize the runtime of the algorithm by caching the resulting DFA states (compute the DFA on the fly), but this requires significant changes to the algorithm and consumes additional space [Cox07].

DFA-based regular expression matchers are commonly used in lexical analysers, where the expressions are known very much in advance, and have relatively simple forms (token specifiers). Moreover, performance of lexical analysers is quite important that it make sense to precompute the DFA in advance. General purpose pattern matchers on the other hand tend not to rely on DFA, as their construction can lead to state space explosion, especially with user generated regular expressions. One might think therefore, that general purpose regular expression matchers must be based on the lock-

step algorithm (or a variant of it). However, it may come as a surprise that most regular expression libraries operate on a backtracking algorithm, a relatively unfamiliar algorithm to the automata world. What is more unsettling is that these backtracking matchers have exponential worst case runtimes, and regular expressions which trigger this behaviour are not at all uncommon. For a straightforward example, consider matching the regular expression: `([A-z]|])*Z` against an input string of the form `]`^{n} (i.e. the character '`]`' repeated n times). The Java Virtual Machine (JVM) grinds to a halt on this task for $n \sim 50$. Understanding this phenomena, assessing its implications and building a sound theoretical formalization of this unorthodox approach to regular expression matching are central topics of the present thesis.

1.2 Backtracking pattern matchers

If we look at the matching problem from a language perspective, we are likely to come up with either a DFA simulation or an NFA simulation, this is what automata theory has taught us over the years. On the other hand, the inductive structure of regular expression syntax gives rise to a much more intuitive semantics. A regular expression is either *empty* (ε), an *atom* (a symbol), a *concatenation* (ee'), an *alternation* ($e|e'$) or a *repetition* (e^*). Figure 1.1 presents a matching routine which exploits this structure. Here we have assumed a functional style of programming (to avoid clutter), although the idea can be easily adopted to an imperative setting. This algorithm operates by backtracking, the `orelse` clauses (conditional function calls)

$$\texttt{match} : \texttt{exp} \to \texttt{string} \to \texttt{bool}$$
$$\texttt{match'} : \texttt{exp} \to (\texttt{string} \to \texttt{bool}) \to \texttt{string} \to \texttt{bool}$$

$$\texttt{match'}\ \varepsilon\ k\ w = k\ w$$
$$\texttt{match'}\ a\ k\ \varepsilon = \texttt{false}$$
$$\texttt{match'}\ a\ k\ bw = \texttt{if}\ b == a\ \texttt{then}\ k\ w\ \texttt{else false}$$
$$\texttt{match'}\ (ee')\ k\ w = \texttt{match'}\ e\ (\texttt{fn}\ w\ \to \texttt{match'}\ e'\ k\ w)\ w$$
$$\texttt{match'}\ (e|e')\ k\ w = \texttt{match'}\ e\ k\ w\ \texttt{orelse match'}\ e'\ k\ w$$
$$\texttt{match'}\ e^*\ k\ w = k\ w\ \texttt{orelse match'}\ (e\,e^*)\ k\ w$$
$$\texttt{match}\ e\ w = \texttt{match'}\ e\ (\texttt{fn}\ w\ \to w == \varepsilon)\ w$$

Figure 1.1: A backtracking pattern matcher

give rise to branch points in the search for a match, these branch points are saved on the stack and re-visited in case of failure.

The simplicity of this algorithm has made it popular as a coding exercise in functional programming courses [Har97, Har99]. More strikingly though, this algorithm has also established itself as the de-facto choice for implementing regular expression matchers. It should be noted that this is not isolated to a couple of implementations; Java, .Net, Perl, PCRE, Python, Javascript and many other frameworks provide regular expression support based on a similar backtracking algorithm [Cox07]. The reason for this choice consists of the following factors:

(a) The backtracking algorithm is quite versatile in that the pattern specification language can be easily extended to support non-regular constructs

(b) Lack of understanding of core computer science concepts among practitioners

We discuss these points under the topics below.

1.2.1 Irregular expressions

A requirement of the backtracking algorithm is that an ordering must be imposed at branch points. In the code listing above, the left alternation of $(e_1|e_2)$ is prioritized over the right one. This effectively renders the alternation operator non-commutative; as the algorithm will always prefer the left branch, even if an easier match is available through the right branch. While backtracking matchers (and their users) have accepted this to be the default behavior for the alternation operator, it may sound quite alien to someone familiar with the usual interpretation of regular expressions.

For the Kleene operator (e^*), the code above has prioritized matching ε over repeating e. In other words, the Kleene operator behaves in a reluctant fashion, matching the least number of e's as possible. The other alternative is to make the Kleene operator greedy, forcing it to match as many copies of e's as possible. For an example, consider matching the regular expression $(a|b)^*b$ against the input string:

```
aaabaaab
```

If the Kleene operator behaves in a reluctant fashion, the prefix *aaab* will produce a successful match. On the other hand, a greedy Kleene operator will yield a match for the entire string. The two interpretations produce quite different results. For this reason, backtracking matchers provide special syntax that allows one to choose between these two interpretations; the Kleene operator may either be reluctant ($e^{*?}$) or greedy (e^* or e^{**}). One may

argue that these Kleene quantifiers represent a useful feature of backtracking matchers, especially since implementing such a semantics on a DFA/NFA based matcher is quite challenging [Cox09].

Backtracking matchers also support *submatch-extraction* [YMH+12], more formally known as *parse-extraction* [Kea91]. The idea here is to extract structured information from input strings; to know which sub-strings matched which sub-expressions. The intent to extract a submatch is expressed by use of special parentheses. For an example, consider the pattern:

$$\texttt{(?P<x>}(a|b)^*\texttt{)}b$$

Where the sub-expression $(a|b)^*$ is marked for submatching (using Python's named capturing groups notation [Fou12]). If this expression were to be matched against the input string:

```
aaabaaab
```

The result of the submatch would be (`x` =) `aaa` for a reluctant Kleene operator or `aaabaaa` for a greedy Kleene operator. Some research has shown that similar features can be adapted to DFA/NFA based matchers, with a considerable amount of programming effort [Cox09, Lau01]. What is important however is that this feature requires so little effort to be implemented on a backtracking matcher. Intuitively, a backtracking matcher holds a single search path at any given point in time, which makes it efficient to keep track of all the submatches corresponding to that derivation.

With submatching comes the possibility of recalling a previous submatch at a later point in the expression. Matching the so called *backreferences* is known to be an NP-hard problem [Aho90], an exhaustive search being the

only viable solution. Recent backtracking matchers have gone even beyond backreferences, making it possible to programmatically modify the submatching behavior at runtime [Cox11]. While these extensions may sound perverse, they make a strong case for the versatility of the backtracking approach for general purpose pattern matching. DFA/NFA based pattern matchers on the other hand can only be pushed to support a very limited set of additional pattern constructs.

1.2.2 Pathological runtimes

Backtracking pattern matchers are straightforward to implement. They offer great flexibility towards supporting non-regular pattern constructs. There is no free lunch however, ensuring termination and avoiding exponential blowups are serious challenges.

The backtracking matcher presented in Figure 1.1 can lead to nontermination; matching the expression ε^* against the input string a will cause this matcher to enter an infinite loop. Fixing such infinite loops in a consistent manner and ensuring termination have proven to be quite challenging, both in theory [Har99, DN01] as well as in practice [Haz12b].

As opposed to non-termination (which can be overcome) however, exponential blowups are an inevitable reality of backtracking pattern matchers. For a simple demonstration, consider matching the regular expression $(a|b|ab)^*bc$ against an input string of the form $(ab)^nac$. Most widespread backtracking pattern matchers (Java, .NET, Python etc.) will become nonresponsive on this task, even for modest input sizes ($n \sim 50$). Such *patho-*

logical patterns [Cox07] are quite common in practice that attacks on them have a special name: REDoS, short for Regular Expression Denial of Service [OWA12, RW12].

REDoS phenomena poses an important research problem due to several reasons. Firstly, one of the primary applications of regexes (i.e. regular or otherwise patterns) is input sanitization, which is used to safeguard systems against other kinds of attacks (command injection, buffer overflows etc.). For an example, Snort [Sou12] is a network packet inspection software that attempts to detect potentially harmful network traffic. It has been shown that poorly crafted Snort detection rules can lead to REDoS attacks, which (ironically) leaves protected systems unavailable [SEJ06]. Secondly, the so called pathological patterns behave as intended for almost all the time, it is only a certain kind of input that can trigger a *catastrophic backtracking* [Goy09a] event. REDoS vulnerabilities can therefore remain undetected for as long as such abnormal input is not received, which makes them even more precarious.

Note that a plain resource limit (CPU time or memory) on the matching routine is not a reliable solution for REDoS as such a limit would depend on heuristics and therefore interfere with the processing of benign input. In [NN10], the authors note that such limits can indeed cause harmless input to be dropped. Moreover, even with an ideal resource limit, an attack could still be mounted by repeatedly invoking the exponential vulnerability and thereby driving each request to the resource limit (e.g. network packet filters such as Snort have to process thousends of packets in parallel, the accumulative effect of multiple packets hitting the upper resource limit in parallel can be quite harmful). Therefore, a reliable REDoS protection

based on resource-limits would require a much more sophisticated strategy than a simple (static) resource limit.

Finally, the REDoS phenomenon is poorly understood at present. Apart from a few internet articles [Cox07, Sul13, OWA12, RW12], there has not been any formal treatment of the REDoS problem up until now. The most well known REDoS analysers at present are limited to visual debugging [Goy09a] (which is only useful for analysing an already observed exponential blowup) and techniques based on fuzzing, where randomly generated strings are matched against the input pattern (in a brute-force manner) with the hope of uncovering exponential blowups. One of the well known fuzzing based analysers is the Microsoft Regex Fuzzer [Mic11], which is a recommended tool for analysing regular expression vulnerabilities during the Microsoft Security Development Lifecycle [Cor12] (a software development process for developing security critical applications). Needless to say, all such tools suffer from the usual deficiencies associated with randomized brute-force techniques (increased detection time, false negatives etc.).

1.3 Contributions of this research

One of the main goals of this research is to develop a semantics that enable reasoning about backtracking pattern matchers. We use operational semantics and abstract machines (from programming language theory) as our primary tools. The discussion is mostly focused on matching regular expressions via backtracking, however, we point out how the developed techniques can be extended to non-regular constructs where appropriate. More

specifically, this research makes the following contributions:

1. A novel semantics based approach for pattern matching is presented, which allows reasoning about NFA based matchers (lockstep) as well as backtracking matchers.
2. The REDoS phenomenon is explored in depth using the machinery developed above, which allows us to develop a static analysis for REDoS. The correctness of the analysis is then established with rigorous mathematical proofs. We also produce an implementation of the analysis in OCaml.
3. Practical usefulness of the analyser is demonstrated by finding real world REDoS vulnerabilities. In comparison to the Microsoft Regex Fuzzer, the tool is shown to operate much more reliably (due to the analysis being sound and complete) and efficiently (orders of magnitude faster).

1.4 Thesis outline

The present chapter served as an introduction to the problem domain. Chapter 2 presents an overview of the research literature relevant to the development of the thesis. In Chapter 3, we develop our semantic view of regular expressions and establish correctness properties of two different regular expression matchers. Chapter 4 elaborates on the exponential runtime blowups and proposes a static analysis for detecting such vulnerabilities. Chapter 5 presents the main theoretical result of this thesis; the soundness and the completeness of the aforementioned static analysis. Chapter 6 presents *RXXR*,

an implementation of our static analysis in OCaml, we also demonstrate its effectiveness in practice by finding real world vulnerabilities. Finally, Chapter 7 concludes the thesis with a discussion of current limitations of the work and possible future research directions.

CHAPTER 2

LITERATURE REVIEW

This chapter presents a review of the background literature most relevant to the present research. Our aim here is to provide a context for the research that is to be developed in the following chapters. We approach this goal by discussing the most relevant material in some detail while pointing to additional literature that branch away from the main topic (that is not greatly relevant to the development of the thesis).

The presentation is divided into four main sections. Section 2.1 provides a condensed introduction to the subject of regular languages, with references to the corresponding literature that is considered to be foundational in nature. In Section 2.2, we discuss the topic of regular expression derivatives; a recurring theme of discussion throughout the thesis. Section 2.3 provides a summary of the recent developments surrounding the subjects of regular expressions and pattern matching. Finally, Section 2.4 discusses those literature from the programming languages (semantics) research from which we have borrowed ideas for the development of the present work.

2.1 Regular languages and automata

A language in computer science is a set of words; words made out of some alphabet (i.e. a set of characters / symbols). For a straightforward example, we may define the language of two character words in the following manner:

$$L_{2c} = \{\text{“}xy\text{”} \mid x, y \in \Sigma\}$$

Here we have assumed the alphabet Σ is given (e.g. $\Sigma = \{a, b, c\}$). Note that the string indicator quotes (“ and ”) are usually omitted in the mathematical notation (unlike in most programming languages).

A language as a set of words view does not itself produce interesting mathematical properties. This is where different classes of languages come into the picture; a class (a particular set) of languages shares a common mathematical formulation. The present classification of all languages into four main classes is due to [Cho56], and is usually depicted as in Figure 2.1. Here the inclusion of one language class inside another signifies that the latter class of languages is more “expressive”, in the sense that the mathematical notation used to construct the more expressive class of languages can capture all of the languages in the less expressive class and more. The present thesis mainly focuses on the regular class of languages (with some discussions on context-sensitivity). Interested readers may refer to literature on the subject of *models of computation* (such as [HU79]) for a general understanding of the different language classes.

Regular languages were first formulated by Kleene in 1956 [Kle56, ALSU07], where he introduced *regular expressions* to denote regular languages:

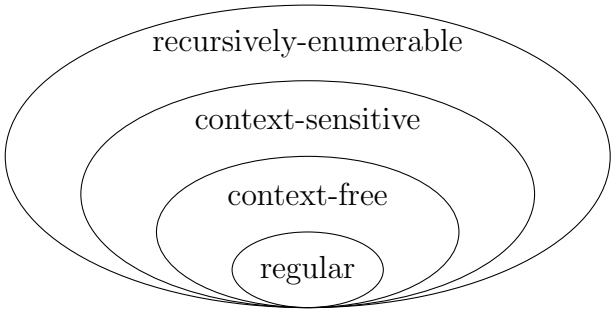


Figure 2.1: The Chomsky hierarchy

Definition 2.1.1 (Regular Expressions [Kle56, MY60]) The language $\mathcal{L}(e)$ denoted by a regular expressions e is defined as follows:

$$\mathcal{L}(\phi) = \{\} \quad \texttt{// empty language}$$

$$\mathcal{L}(\varepsilon) = \{\varepsilon\} \quad \texttt{// empty string}$$

$$\mathcal{L}(a) = \{a\}\ (a \in \Sigma) \quad \texttt{// literal}$$

$$\mathcal{L}(e_1 e_2) = \{xy \mid x \in \mathcal{L}(e_1) \wedge y \in \mathcal{L}(e_2)\}$$

$$\mathcal{L}(e_1 \mid e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$$

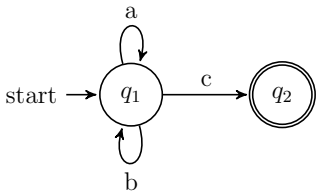
$$\mathcal{L}(e^*) = \{\varepsilon\} \cup \mathcal{L}(ee^*)$$

The definition follows a case analysis of e's syntactic structure (note that the repetition operator $*$ is commonly referred to as the Kleene star). Regular languages are important because they represent the class of languages that is identified by finite automata (state machines), a large portion of fundamental problems in computer science can be effectively solved with these devices (string tokenizing, text search etc.). In fact, the above definition of regular expressions was initially conceived by Kleene [Kle56] as a way of representing events in a Nerve Net (which can be simplified to a finite automaton).

The mathematical representation of a finite automaton is usually a 5-tuple:

$$\langle Q, \Sigma, \delta, q_0, F \rangle$$

Where Q is a set of finite states, Σ an input alphabet, δ a transition function, $q_0 \in Q$ the initial state and $F \subseteq Q$ is the set of final states. The following automaton for example accepts the regular language $(a \mid b)^* c$:



We can represent this automaton with the following 5-tuple:

$$\langle \{q_1, q_2\}, \{a, b, c\}, \delta', q_1, \{q_2\} \rangle$$

$$\delta' = \{((q_1, a), q_1), ((q_1, b), q_1), ((q_1, c), q_2)\}$$

Where δ' is a mapping of $Q \times \Sigma$ pairs to Q. Note that we have intentionally left out the failure transitions (for brevity); transitions not available in δ' (e.g. (q_2, b)) lead to a common failure state (an implicit collector state with no outgoing transitions).

The example above is a Deterministic Finite Automaton (DFA), where all the transitions of the state machine are deterministic. Therefore, the transition function takes the form:

$$\delta : (Q \times \Sigma) \to Q$$

Non-deterministic Finite Automata (NFA) on the other hand can contain

non-deterministic transitions, which gives rise to the transition function:

$$\delta : (Q \times \Sigma) \rightarrow \mathcal{P}(Q)$$

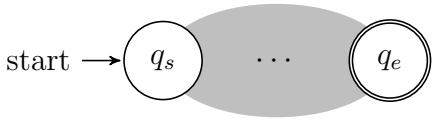
A third form of automata (ε-NFA) also allows silent transitions; transitions that take place without consuming input symbols:

$$\delta : (Q \times \Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$$

These are the most common variants of automata found in literature. While all three forms are known to have equal expressive power [ALSU07, HU79], they differ in space requirements and simulation (runtime) complexities. We will briefly discuss these aspects under the following topics.

2.1.1 Automata construction

A number of automata construction algorithms are known, a detailed survey is presented in [Wat94]. In the present thesis we are especially interested in one of these constructions, initially proposed by Ken Thompson [Tho68]. Thompson's construction follows the inductive structure of the regular expression syntax. For each regular expression e, it generates a state graph of the form:



Where q_s and q_e represent the initial and the final (accepting) states of the overall automaton. This recursive construction is illustrated in Figure 2.2. It is important to observe here that the number of states in the resulting automata is linear in the size of the input expression.

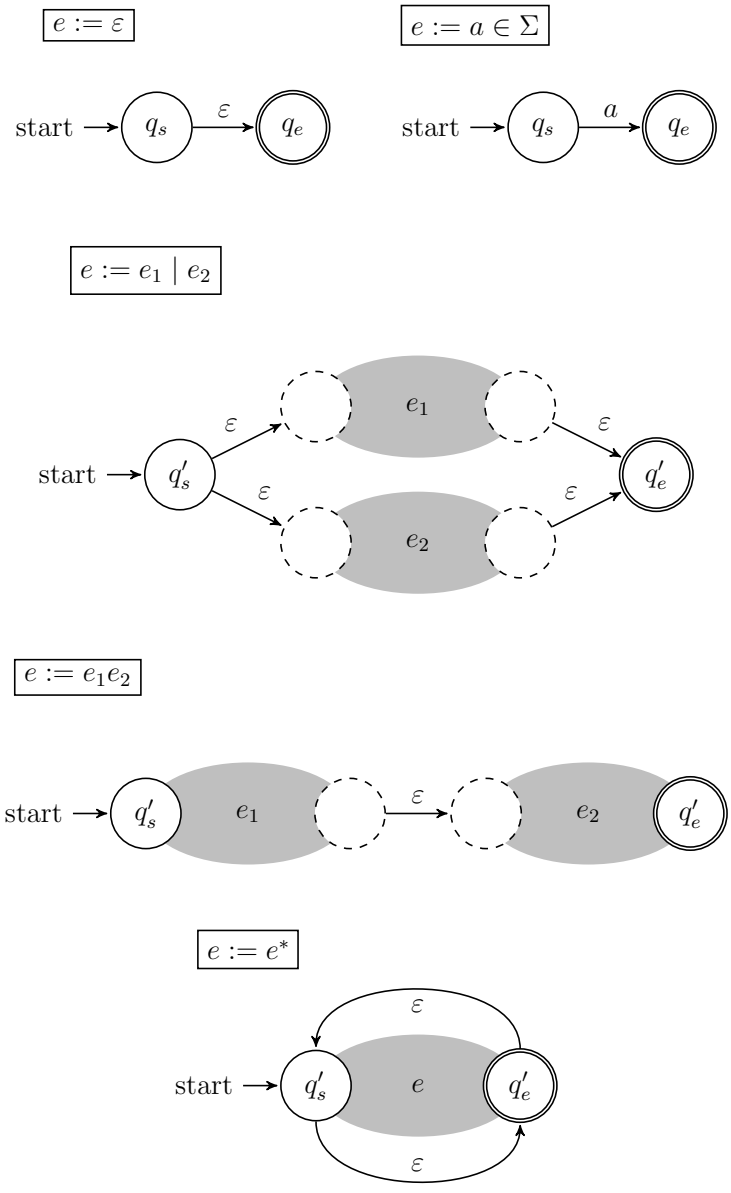


Figure 2.2: Thompson's construction illustrated

Thompson's algorithm has become a common choice among automata implementations due to its simple inductive nature. However, the non-deterministic nature of the resulting ε-NFA means that its simulation is not as straightforward as with DFAs (which can be simulated in linear time). One option is to convert the ε-NFA into a DFA using a procedure known as subset construction [ALSU07]. The idea behind subset construction can be understood as follows: Let e be a regular expression with a corresponding NFA N_e, simulating this NFA against input string w results in a set of states:

$$w \vdash N_e : \{q_s\} \rightarrow Q'$$

Assume D_e is a DFA corresponding to e, its simulation against w would be of the following form:

$$w \vdash D_e : q_s \rightarrow q'$$

This suggests that each state of D_e is simulated by a subset of states in N_e. Thus, D_e can be constructed by exploring N_e and computing all possible subsets of states reachable from $\{q_s\}$. The downside of this transformation however is that in the worst case, all the subsets of states in N_e might be reachable from $\{q_s\}$, resulting in an exponentially large DFA. This is a fundamental limitation of DFAs [ALSU07]. The problem is quite pronounced (and well known) especially since regular expressions causing such blowups are not uncommon in practice [YCD+06].

An alternative option is to remove some of the non-determinism resulting from Thompson's construction (without constructing a DFA). For an example, an ε-closure computation [ALSU07] can be used to remove all the ε transitions to yield an ε-free NFA. The general problem of minimizing an

NFA however is known to have no efficient solution [GS07].

Finally, it should be noted that Definition 2.1.1 is restricted [MY60] in the sense that it does not permit arbitrary logical operators such as negation, intersection or language difference. While such syntactic extensions are desirable in some situations [ORT09], they do not extend the expressive power of regular expressions (i.e. Definition 2.1.1 covers the whole class of regular languages). Moreover, these extensions do not translate to simple NFA building-blocks like other operators in the Thompson's NFA construction. They require additional tweaks in the generated automaton [MY60], leading to complex NFA construction procedures. For this reason, such extensions have fallen out of favor among mainstream regular expression libraries.

2.1.2 Automata simulation

Simulating a DFA is straightforward. On the other hand, there are several algorithms to simulate an NFA. In his paper [Tho68], Thompson also introduced an NFA simulation algorithm; presently known as the lockstep simulation [Cox07]. Instead of a single active state (as with the DFA simulation), the lockstep algorithm computes the set of states reachable from the initial state, for the current prefix of the input string. The algorithm is identical to the subset construction discussed earlier. However, at each step the computed DFA state (set of NFA states) is only kept in memory until the next set of active states is computed. The resulting simulation has linear runtime complexity.

Russ Cox suggests a hybrid simulation [Cox07] (initially used in `egrep`

[ALSU07]) which dynamically constructs a DFA alongside the usual lockstep simulation. Instead of discarding the computed DFA states, a partial DFA is maintained in memory (subject to space constraints). The resulting algorithm is shown to be more efficient than the plain lockstep simulation [Cox10] (only second to pure DFA simulation).

A more naive approach to NFA simulation is to perform a depth-first search of the NFA. While being far inferior in efficiency compared to other alternatives, backtracking has become the most common algorithm used for matching regular expressions [Cox07]. The reasons for this quite surprising observation and the implications of the backtracking paradigm are two main themes of discussion in the present thesis.

2.2 Regular expression derivatives

Regular languages have mostly been studied with finite automata as the underlying mathematical model. On the other hand, Brzozowski's work on regular expression derivatives [Brz64] has a more syntax-oriented flavor - an appealing property from a programming languages point of view. Brzozowski initially established the connection between derivatives and deterministic finite automata, where he developed a DFA construction algorithm which allows regular expressions to contain arbitrary logical connectives (a feat not easily achieved with traditional automata, as mentioned earlier).

Here we present a brief overview of regular expression derivatives. We will relate to these ideas from our own research in the coming chapters.

Definition 2.2.1 Given a language L and a string w, the derivative of L

with respect to w is defined as:

$$\mathcal{D}_w(L) = \{w' | ww' \in L\}$$

Definition 2.2.2 For a language L, the function $\delta : \mathtt{L} \rightarrow \mathtt{exp}$ is defined as:

$$\delta(L) = \begin{cases} \varepsilon & \varepsilon \in L \\ \phi & \varepsilon \notin L \end{cases}$$

Where ϕ is a regular expression constant corresponding to the empty language. The definition of δ is also extended for regular expressions (with a recursive definition on the structure of e).

Definition 2.2.3 The derivative of regular expression e with respect to input symbol a, $\partial_a(e)$ is inductively defined as:

$$\begin{aligned}
\partial_a(\phi) &= \phi \\
\partial_a(\varepsilon) &= \phi \\
\partial_a(a) &= \varepsilon \\
\partial_a(b) &= \phi \quad (b \neq a) \\
\partial_a(e_1\, e_2) &= \partial_a(e_1)\, e_2 \mid \delta(e_1)\, \partial_a(e_2) \\
\partial_a(e_1 \mid e_2) &= \partial_a(e_1) \mid \partial_a(e_2) \\
\partial_a(e^*) &= \partial_a(e)\, e^*
\end{aligned}$$

Theorem 2.2.4 For a regular expression e and an input symbol a ($a \in \Sigma$), the following holds:

$$\mathcal{D}_a(\mathcal{L}(e)) = L(\partial_a(e))$$

This result is quite remarkable in that it allows one to calculate the derivative of a regular language by simply calculating the derivative of the corre-

sponding regular expression. Symbol derivatives of regular expressions are further extended to word derivatives as follows:

$$\partial_{au}(e) = \partial_u(\partial_a(e))$$

For completeness, the empty derivative is defined as $\partial_\varepsilon(e) = e$.

Definition 2.2.5 Two regular expressions that denote the same regular language (but not necessarily identical in form) are said to be of the same *type*.

Lemma 2.2.6 The derivative $\partial_w(e)$ of a regular expression e with respect to a string w is also a regular expression.

Theorem 2.2.7 A string w is contained in a regular language $\mathcal{L}(e)$ if and only if ε is contained in $\partial_w(e)$.

Theorem 2.2.8 (a) Every regular expression e has a finite number d_e of types of derivatives. (b) At least one derivative of each type must be found among the derivatives with respect to strings of length $d_e - 1$.

Based on Theorem 2.2.8, Brzozowski illustrates a procedure for building a DFA from a given regular expression. The strings of the input alphabet are enumerated while calculating the derivatives of the regular expression at hand. Each new derivative is compared against the current set of derivatives to determine if it belongs to a type of derivative already discovered. The process terminates due to Theorem 2.2.8 (b).

As an example, let us consider the Brozozowsky DFA construction for the regular expression $(ab \mid b)^*ba$. First we observe that this regular expression only utilises the symbols a and b; so that we can limit our derivations to those

symbols only (any other derivative would be ϕ). The first two derivatives of the initial expression $(ab \mid b)^*ba$ are illustrated below:

$$\begin{aligned}
\partial_a((ab \mid b)^*ba) &= \partial_a((ab \mid b)^*)\, ba \mid \delta((ab \mid b)^*)\, \partial_a(ba) \\
&= \partial_a((ab \mid b))\, (ab \mid b)^* ba \mid \varepsilon\, \phi \\
&= (\partial_a(ab) \mid \partial_a(b))\, (ab \mid b)^* ba \\
&= b\, (ab \mid b)^* ba \qquad (e_1)
\end{aligned}$$

$$\begin{aligned}
\partial_b((ab \mid b)^*ba) &= \partial_b((ab \mid b)^*)\, ba \mid \delta((ab \mid b)^*)\, \partial_b(ba) \\
&= \partial_b((ab \mid b))\, (ab \mid b)^* ba \mid \varepsilon\, a \\
&= (\partial_b(ab) \mid \partial_b(b))\, (ab \mid b)^* ba|a \\
&= (ab \mid b)^* ba|a \qquad (e_2)
\end{aligned}$$

That is, the first two derivatives are unique. The computation continues with these new derivatives until all the types of derivatives are discovered. We can see straightaway that $\partial_b(e_1)$ yields back the original expression and $\partial_a(e_1)$ is ϕ. The entire operation is visualized in Figure 2.3 (taken from [BS86]). The final DFA is obtained by turning each type of derivative into a DFA state and each derivation ($\partial_a(\ )$) into a transition ($\xrightarrow{a}$).

Like subset construction, DFA construction by derivatives leads to exponential blowups. Derivatives of linear regular expressions [BS86] and partial derivatives [Ant96] have been proposed to relax some aspects of Brzozowski derivatives, allowing the construction of an intermediate (ε-free) NFA prior to the DFA conversion. Note however that these constructions do not support regular expressions with arbitrary logical connectives. Thompson [Tho68] notes that the lockstep simulation can be viewed as caclulating the deriva-

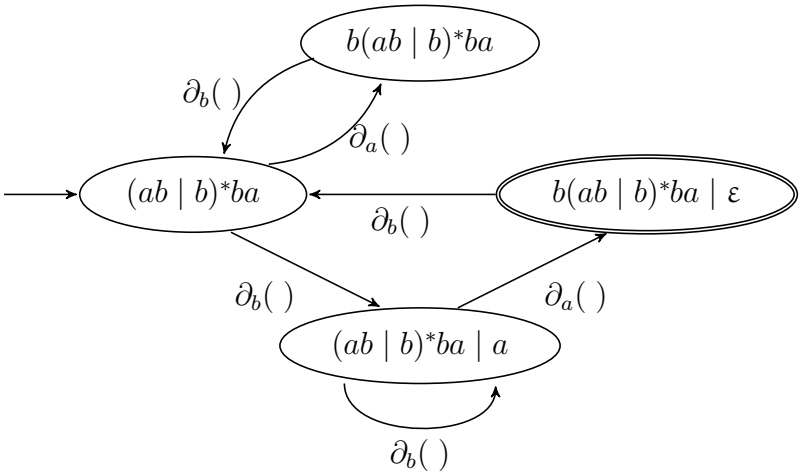


Figure 2.3: A DFA accepting $(ab|b)^*ba$

tives of the regular expression on-the-fly; in this sense the set of active NFA states (of the lockstep simulation) can be viewed as a collection of partial derivatives, which forms the current Brzozowski derivative. Finally, canonical derivatives have been proposed [CZ02] to draw the connections between these different types of derivatives. There the authors also describe how canonical derivatives can be used to devise an even more efficient NFA construction.

2.3 Recent developments

In a series of articles [Cox07, Cox09, Cox10], Russ Cox presents the modern day mismatch between theory and practice of regular expressions. Cox exposes the weaknesses of the backtracking approach to regular expression matching and compares that to the Thompson's lockstep algorithm [Tho68]. The relative flexibility of the backtracking paradigm however seems to have received less attention in this exposition, the Perl community in particular

has disagreed with this treatment [Per07] and holds the view that efficiency needs to be balanced with other aspects of developing large software systems [Per05].

While regular expressions and automata theory emerged as success stories of computer science, Brzozowski's derivatives [Brz64] seem to have been forgotten over time [ORT09]. A refresher on this old technique is presented in [ORT09]. The authors share their experience in building practical regular expression matchers based on derivatives while highlighting the added flexibility of using boolean operators in regular expression specifications.

Aside from boolean operators (due to derivatives), other recent studies have focused on even more powerful (sometimes irregular) pattern constructs. For an example, submatching (sometimes known as parse extraction [FC04]) attempts to capture the sub-strings matched by the individual sub-expressions. Laurikari [Lau01] presents an augmented form of automata (tagged automata) to solve this problem while [Sul12] employs a variant of Brzozowski's derivatives. Interest on even more expressive pattern constructs has been growing in the past decade [BC08, Cox11].

Finally, there has been some interest in exploiting the parallelism in NFAs with GPGPU computing [NVI11]. An implementation based on our own research is discussed in [RT11]. A more practical (and more efficient) implementation is presented in [CRRS10].

2.4 Programming languages research

The work presented in this dissertation was inspired and shaped by programming languages theory. We will briefly discuss these literature under the following topics.

2.4.1 Semantics

Semantics describe the meaning of programs, which can be stated in many forms. Some of the commonly used variants include *Operational semantics*, *Denotations semantics* and *Axiomatic semantics* [Pie02, Rey99]. In the present work, we use operational semantics, in the forms of big-step semantics [Pie02] and abstract machines.

2.4.2 Abstract machines

Abstract machines are the theoretical counterpart of virtual machines. While a virtual machine defines a set of instructions (like bytecodes in the Java Virtual Machine) and an execution environment, an abstract machine defines a mathematical machine configuration and a set of transitions. The following illustration shows an idealized abstract machine:

$$E \vdash \langle C_1, C_2, \ldots, C_n \rangle \rightarrow \langle C'_1, C'_2, \ldots, C'_n \rangle$$

Here the machine is operating under the environment E and it has n components. Usually an abstract machine consists of a set of transitions, which are defined based on possible machine configurations. The SECD machine (named after its components) was introduced in 1964 by Landin [Lan64] as a

(theoretical) interpreter for the lambda calculus, which was later refined into the CEK machine [FF86]. The WAM [AK91] is a complex abstract machine capable of modeling the execution of logic programs. An analysis of the most commonly used features of abstract machine design is presented in [Rey72]. The present work introduces several abstract machines [RT11] for matching regular expressions.

2.4.3 Continuations

The continuation passing style (CPS) is a way of program representation in which functions never return. The relationship between traditional function calls, control statements and explicit continuations is discussed in [Thi99]. Continuations enable programmers to encode complex control flow paths not easily achievable in a procedural environment. They have been used to embed logic programming constructs into functional programming environments [Hay87] and to develop powerful optimizing compilers for higher-order programming languages [App92]. In this work, some of our abstract machines employ explicit continuations to model the behavior of practical regular expression matchers.

2.4.4 Program analysis

Program analysis is a broad research topic. In this work, we are primarily interested in correctness analysis and security analysis. Some of the well developed techniques for reasoning about program correctness include Hoare logic [Hoa69] and later extensions such as separation logic [Rey02, IO01] for

reasoning about pointer structures.

In Chapter 3, we establish the correctness and termination of two different regular expression matchers using abstract machines. The termination of backtracking regular expression matchers has been previously studied in [Har99], where the author demonstrates the process of proof development and how it aids in de-bugging the underlying program. Harper suggests a regular expression re-writing technique which fixes the non-termination of naive backtracking implementations [Har97]. In [DN01], the same problem is used as an example in a study of defunctionalization [Rey72], where the authors compare the correctness proofs of a higher-order regular expression matcher and its defunctionalized (first-order) counterpart. The proof techniques used in these three works are quite different from one another, we will discuss these differences later in the thesis.

Security analysis

Software analysis for security is by now a well established discipline in software engineering [DMS06]. The kinds of security vulnerabilities analysed range from SQL injection attacks in web applications [HVO06] to buffer overflows in system software [SK02].

While there are quite a lot of approaches to software security (input sanitization, load balancing, logging, crash recovery, manual checking etc.), the more sophisticated techniques employ static analysis for vulnerability detection [LL05, CM04, SK02]. Note that the term *static* suggests an analysis taking place prior to the deployment of the program, it does not suggest the non-execution of the source program. Certain static analysis techniques

involve the symbolic execution of the program in question [CF10], abstract interpretation [Cou13] goes even further where all possible execution paths are analysed with respect to some abstraction.

CHAPTER 3

ABSTRACT MACHINES FOR PATTERN MATCHING

In this chapter, we formalize the view of regular expression matchers as machines by using tools from programming language theory, specifically operational semantics. We do so starting from the usual definition of regular expressions and their meaning, and then defining increasingly realistic machines.

We first define some preliminaries and recall what it means for a string to match a regular expression in Section 3.1; from our perspective, matching is a simple form of big-step semantics, and we aim to refine it into a small-step semantics. To do so in Section 3.2, we introduce a distinction between a current expression and its continuation. We then refine this semantics by representing the regular expression as a syntax tree using pointers in memory (Section 3.3). Crucially, the pointer representation allows us to compare sub-expressions by pointer equality (rather than structurally). This pointer equality test is needed for the efficient elimination of redundant match attempts, which underlies the general lockstep NFA simulation presented in Section 3.3.1.

$\boxed{e \downarrow w}$

$$\frac{e_1 \downarrow w_1 \qquad e_2 \downarrow w_2}{(e_1\, e_2) \downarrow (w_1\, w_2)}\ (\textsc{Seq}) \qquad \frac{}{a \downarrow a}\ (\textsc{Match}) \qquad \frac{}{\varepsilon \downarrow \varepsilon}\ (\textsc{Epsilon})$$

$$\frac{e \downarrow w_1 \qquad e^* \downarrow w_2}{e^* \downarrow (w_1\, w_2)}\ (\textsc{Kleene1}) \qquad \frac{}{e^* \downarrow \varepsilon}\ (\textsc{Kleene2})$$

$$\frac{e_1 \downarrow w}{(e_1 \mid e_2) \downarrow w}\ (\textsc{Alt1}) \qquad \frac{e_2 \downarrow w}{(e_1 \mid e_2) \downarrow w}\ (\textsc{Alt2})$$

Figure 3.1: Regular expression matching as a big-step semantics

3.1 Regular expression matching as a big-step semantics

Let Σ be a finite set, regarded as the input alphabet. For regular expressions we use the abstract syntax introduced in Definition 1.1.1, except for the empty expression ϕ which we ignore from here onwards (this avoids clutter and has no impact on the results).

We let e range over regular expressions, a over characters, and w over strings of characters. The empty string is written as ε. Note that there is also a regular expression constant $\mathfrak{e}$. We also write the sequential composition $e_1\, e_2$ as $e_1 \bullet e_2$ when we want to emphasize it as the occurrence of an operator applied to e_1 and e_2, for instance in a syntax tree. For strings w_1 and w_2, we write their concatenation as juxtaposition $w_1 w_2$. A single character a is also regarded as a string of length 1.

Our starting point is the usual definition of what it means for a string w to match a regular expression e. We write this relation as $e \downarrow w$, regarding it as a big-step operational semantics for a language with non-deterministic branching $e_1 \mid e_2$ and a non-deterministic loop e^*. The rules are given in

Figure 3.1.

Some of our operational semantics will use lists. We write $h :: t$ for constructing a list with head h and tail t. The concatenation of two lists s and t is also written as $s :: t$ (so the operator $::$ is overloaded). For example, $1 :: [2] = [1, 2]$ and $[1, 2] :: [3] = [1, 2, 3]$. The empty list is written as $[\,]$.

3.2 The EKW and EKWF machines

The big-step operational semantics of matching in Figure 3.1 gives us little information about how we should attempt to match a given input string w. We define a small-step semantics, called the EKW machine, that makes the matching process more explicit. In the tradition of the SECD machine [Lan64], the machine is named after its components: E for expression, K for continuation, W for word to be matched.

Definition 3.2.1 A configuration of the EKW machine is of the form $\langle e \ ; \ k \ ; \ w \rangle$ where e is a regular expression, k is a list of regular expressions, and w is a string. The transitions of the EKW machine are given in Figure 3.2. The accepting configuration is $\langle \varepsilon \ ; \ [\,] \ ; \ \varepsilon \rangle$.

Here e is the regular expression the machine is currently focusing on. What remains to the right of the current expression is represented by k, the current continuation. The combination of e and k together is attempting to match w, the current input string.

Note that many of the rules are fairly standard, specifically the pushing and popping of the continuation stack. The machine is non-deterministic.

$\boxed{\langle e \;;\; k \;;\; w\rangle \to \langle e' \;;\; k' \;;\; w'\rangle}$

$$\langle e_1 \mid e_2 \;;\; k \;;\; w\rangle \xrightarrow{\texttt{alt1}} \langle e_1 \;;\; k \;;\; w\rangle \tag{3.1}$$

$$\langle e_1 \mid e_2 \;;\; k \;;\; w\rangle \xrightarrow{\texttt{alt2}} \langle e_2 \;;\; k \;;\; w\rangle \tag{3.2}$$

$$\langle e_1\, e_2 \;;\; k \;;\; w\rangle \xrightarrow{\texttt{conc}} \langle e_1 \;;\; e_2 :: k \;;\; w\rangle \tag{3.3}$$

$$\langle e^* \;;\; k \;;\; w\rangle \xrightarrow{\texttt{kln1}} \langle e \;;\; e^* :: k \;;\; w\rangle \tag{3.4}$$

$$\langle e^* \;;\; k \;;\; w\rangle \xrightarrow{\texttt{kln2}} \langle \varepsilon \;;\; k \;;\; w\rangle \tag{3.5}$$

$$\langle a \;;\; k \;;\; a\, w\rangle \xrightarrow{\texttt{match}} \langle \varepsilon \;;\; k \;;\; w\rangle \tag{3.6}$$

$$\langle \varepsilon \;;\; e :: k \;;\; w\rangle \xrightarrow{\texttt{pop}} \langle e \;;\; k \;;\; w\rangle \tag{3.7}$$

Figure 3.2: EKW machine transition steps

The paired rules with the same current expressions e^* or $(e_1 \mid e_2)$ give rise to branching in order to search for matches, where it is sufficient that one of the branches succeeds.

Lemma 3.2.2 If there is a run of the EKW machine of the form:

$$\langle e \;;\; k \;;\; w\rangle \xrightarrow{*} \langle e' \;;\; k' \;;\; w'\rangle$$

then for any $\bar{k}$ and $\bar{w}$, there is also a run:

$$\langle e \;;\; k :: \bar{k} \;;\; w\bar{w}\rangle \xrightarrow{*} \langle e' \;;\; k' :: \bar{k} \;;\; w'\bar{w}\rangle$$

Proof Observe that each transition of the form:

$$\langle e \;;\; k \;;\; w\rangle \to \langle e' \;;\; k' \;;\; w'\rangle$$

Can be simulated on the extended machine with:

$$\langle e \;;\; k :: \bar{k} \;;\; w\bar{w}\rangle \to \langle e' \;;\; k' :: \bar{k} \;;\; w'\bar{w}\rangle$$

With this result, an induction over the length of the run completes the proof.

□

Lemma 3.2.3 If $e \downarrow w$ then there is a run

$$\langle e \; ; \; [\,] \; ; \; w \rangle \xrightarrow{*} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Proof By induction on the height of the derivation $e \downarrow w$. The applicable base cases are (derivations of height 1):

$$\frac{}{\varepsilon \downarrow \varepsilon} \quad \frac{}{a \downarrow a} \quad \frac{}{e^* \downarrow \varepsilon}$$

Observe that the result holds for these cases from the definitions of the EKW transitions in Figure 3.2. For the inductive step, we perform a case analysis of the structure of e:

- **Empty** ($e = \varepsilon$) **/ Literal** ($e = a$): Already covered (base cases).
- **Alternation** ($e = e_1 \mid e_2$): Without loss of generality, suppose:

  $$\frac{e_1 \downarrow w}{(e_1 \mid e_2) \downarrow w}$$

  Then from the induction hypothesis, we have:

  $$\langle e_1 \; ; \; [\,] \; ; \; w \rangle \xrightarrow{*} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

  Moreover, from the EKW transitions we get:

  $$\langle e_1 \mid e_2 \; ; \; [\,] \; ; \; w \rangle \xrightarrow{\texttt{alt1}} \langle e_1 \; ; \; [\,] \; ; \; w \rangle$$

  These two runs when combined, establishes the desired result.

- **Concatenation** ($e = e_1 e_2$):

  $$\frac{e_1 \downarrow w_1 \quad e_2 \downarrow w_2}{e_1 e_2 \downarrow w_1 w_2}$$

Again from the induction hypothesis, we get:

$$\langle e_1 \; ; \; [\,] \; ; \; w_1 \rangle \xrightarrow{*} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle \tag{A}$$

$$\langle e_2 \; ; \; [\,] \; ; \; w_2 \rangle \xrightarrow{*} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle \tag{B}$$

The EKW transitions (Figure 3.2) yield:

$$\langle e_1 e_2 \; ; \; [\,] \; ; \; w_1 w_2 \rangle \xrightarrow{\texttt{conc}} \langle e_1 \; ; \; e_2 \; ; \; w_1 w_2 \rangle$$

Applying Lemma 3.2.2 to (A) above with $\bar{k} = [e_2], \bar{w} = w_2$ gives:

$$\langle e_1 \; ; \; e_2 \; ; \; w_1 w_2 \rangle \xrightarrow{*} \langle \varepsilon \; ; \; e_2 \; ; \; w_2 \rangle \tag{C}$$

From (C) and (B) we derive:

$$\langle \varepsilon \; ; \; e_2 \; ; \; w_2 \rangle \xrightarrow{\texttt{pop}} \langle e_2 \; ; \; [\,] \; ; \; w_2 \rangle \xrightarrow{*} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

That is, we have shown:

$$\langle e_1 e_2 \; ; \; [\,] \; ; \; w_1 w_2 \rangle \xrightarrow{*} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

- **Kleene** ($e = e_1{}^*$): Follows a similar line of argument to that of concatenation.

□

Lemma 3.2.4 The following equivalences hold true for regular expressions:

$$(e_1 e_2) e_3 \equiv e_1 (e_2 e_3)$$

$$(e_1 \mid e_2) e_3 \equiv (e_1 e_3) \mid (e_2 e_3)$$

$$e_3 (e_1 \mid e_2) \equiv (e_3 e_1) \mid (e_3 e_2)$$

$$e \, \varepsilon \equiv e \equiv \varepsilon \, e$$

$$e^* \equiv (\varepsilon \mid e e^*)$$

Proof Follows from the big-step semantics in Definition 3.1. □

Definition 3.2.5 For a continuation k, we define $\flat k$:

$$\begin{aligned} \flat[\,] &= \varepsilon \\ \flat(e :: k) &= e \flat k \end{aligned}$$

Lemma 3.2.6 If there is a run

$$\langle e \,;\, k \,;\, w \rangle \xrightarrow{n} \langle \varepsilon \,;\, [\,] \,;\, \varepsilon \rangle$$

then $(e \flat k) \downarrow w$.

Proof By induction over the length of the EKW run. Observe that the base case ($n = 0$) trivially holds given that $\varepsilon \downarrow \varepsilon$. For the inductive step, we perform a case analysis of the structure of e.

- **Empty** ($e = \varepsilon$):

  $$\langle \varepsilon \,;\, e :: k \,;\, w \rangle \xrightarrow{\texttt{pop}} \langle e \,;\, k \,;\, w \rangle \xrightarrow{n-1} \langle \varepsilon \,;\, [\,] \,;\, \varepsilon \rangle$$

  From the induction hypothesis we have $(e \flat k) \downarrow w$, which implies $(\varepsilon \flat (e :: k)) \downarrow w$ following the definition of $\flat$ and Lemma 3.2.4.

- **Literal** ($e = a$):

  $$\langle a \,;\, k \,;\, aw \rangle \xrightarrow{\texttt{match}} \langle \varepsilon \,;\, k \,;\, w \rangle \xrightarrow{n-1} \langle \varepsilon \,;\, [\,] \,;\, \varepsilon \rangle$$

  The induction hypothesis yields $(\varepsilon \flat k) \downarrow w$. Given that $a \downarrow a$, the big-step semantics for concatenation (Figure 3.1) gives:

  $$a(\varepsilon \flat k) \downarrow aw$$

  Which in turn reduces to $(a \flat k) \downarrow aw$ (Lemma 3.2.4).

- **Alternation** ($e = e_1 \mid e_2$): The two possible runs are:

$$\langle e_1 \mid e_2 \;;\; k \;;\; w\rangle \xrightarrow{\texttt{alt1}} \langle e_1 \;;\; k \;;\; w\rangle \xrightarrow{n-1} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon\rangle$$

$$\langle e_1 \mid e_2 \;;\; k \;;\; w\rangle \xrightarrow{\texttt{alt1}} \langle e_2 \;;\; k \;;\; w\rangle \xrightarrow{n-1} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon\rangle$$

  From the induction hypothesis we get:

$$(e_1 \flat k) \downarrow w \quad \texttt{or} \quad (e_2 \flat k) \downarrow w$$

  In either case, we have $(e_1 \flat k \mid e_2 \flat k) \downarrow w$. Following Lemma 3.2.4, this implies $(e_1 \mid e_2) \flat k \downarrow w$.

- **Concatenation** ($e = e_1 e_2$):

$$\langle e_1 e_2 \;;\; k \;;\; w\rangle \xrightarrow{\texttt{conc}} \langle e_1 \;;\; e_2 :: k \;;\; w\rangle \xrightarrow{n-1} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon\rangle$$

  The induction hypothesis yields $e_1 \flat (e_2 :: k) \downarrow w$, which reduces to $(e_1 e_2 \flat k) \downarrow w$ following the definition of $\flat$ and Lemma 3.2.4.

- **Kleene** ($e = {e_1}^*$): The two possible runs are:

$$\langle {e_1}^* \;;\; k \;;\; w\rangle \xrightarrow{\texttt{kln1}} \langle \varepsilon \;;\; k \;;\; w\rangle \xrightarrow{n-1} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon\rangle$$

$$\langle {e_1}^* \;;\; k \;;\; w\rangle \xrightarrow{\texttt{kln2}} \langle e_1 \;;\; {e_1}^* :: k \;;\; w\rangle \xrightarrow{n-1} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon\rangle$$

  From the induction hypothesis we get:

$$(\varepsilon \flat k) \downarrow w \quad \texttt{or} \quad (e_1 \flat ({e_1}^* :: k)) \downarrow w$$

  Following the definition of $\flat$ and Lemma 3.2.4, this reduces to:

$$(\varepsilon \flat k) \downarrow w \quad \texttt{or} \quad (e_1 {e_1}^* \flat k) \downarrow w$$

  In either case, we have $(\varepsilon \flat k \mid e_1 {e_1}^* \flat k) \downarrow w$. Which in turn (Lemma 3.2.4) reduces to $(\varepsilon \mid e_1 {e_1}^*) \flat k \downarrow w$ or ${e_1}^* \flat k \downarrow w$.

□

Lemma 3.2.7 (EKW correctness) $e \downarrow w$ if and only if there is a run:

$$\langle e \ ; [\,] \ ; w \rangle \xrightarrow{*} \langle \varepsilon \ ; [\,] \ ; \varepsilon \rangle$$

Proof Follows from Lemma 3.2.3 and 3.2.6. □

Definition 3.2.8 A configuration of the EKWF machine is a list of configurations of EKW machines. The transitions of the EKWF machine are given in Figure 3.3. Given a regular expression e and an initial input w, the initial configuration of the EKWF machine is:

$$\langle e \ ; [\,] \ ; w \rangle :: [\,]$$

The EKWF machine may terminate in one of two ways:

- The list may become empty. In this case, the attempt to match has failed, and the initial input is rejected.
- The configuration is of the form $\langle \varepsilon \ ; [\,] \ ; \varepsilon \rangle :: f$ for some f. In this case, the match is successful, and the initial input is accepted.

The EKWF machine can be thought of as a deterministic version of the EKW machine. Whenever the EKW machine makes a non-deterministic choice, the EKWF machine selects one of those choices and saves the other choice on top of the failure continuation stack. The idea is that if the current EKW machine fails to find a match, the EKWF machine will eventually backtrack to the previously saved EKW instance and continue from there onward.

Lemma 3.2.9 If there is an EKWF run:

$$\langle e \ ; k \ ; w \rangle :: [\,] \xrightarrow{n} \langle e' \ ; k' \ ; w' \rangle :: f$$

Then there is a corresponding EKW run:

$$\langle e \; ; \; k \; ; \; w \rangle \xrightarrow{*} \langle e' \; ; \; k' \; ; \; w' \rangle$$

Proof By induction on the length of the EKWF run (n). For the base case ($n = 1$), we must establish the result for all EKWF transitions except `rej` in Figure 3.3 (`rej` is not a base case for the above EKWF run). This is straightforward since for each of those transitions, there is a corresponding (non-deterministic) EKW transition in Figure 3.2 with the required property. For an example, take the EKWF transition `kln` in Figure 3.3 (with $f = [\,]$):

$$\langle e^* \; ; \; k \; ; \; w \rangle :: [\,] \xrightarrow{\texttt{kln}} \langle \varepsilon \; ; \; k \; ; \; w \rangle :: \langle e \; ; \; e^* :: k \; ; \; w \rangle :: [\,]$$

The corresponding EKW transition in Figure 3.2 is:

$$\langle e^* \; ; \; k \; ; \; w \rangle \xrightarrow{\texttt{kln2}} \langle \varepsilon \; ; \; k \; ; \; w \rangle$$

For the inductive step, we perform a case analysis on the n^{th} step taken by the EKWF machine. Observe that the n^{th} step must be of one of the three possible forms below:

$$\mathring{m} :: [\,] \xrightarrow{n-1} m :: f \to m' :: f \quad (\texttt{conc}, \texttt{match}, \texttt{pop}) \tag{A}$$

$$\mathring{m} :: [\,] \xrightarrow{n-1} m :: f \to m' :: m'' :: f \quad (\texttt{alt}, \texttt{kln}) \tag{B}$$

$$\mathring{m} :: [\,] \xrightarrow{n-1} m :: m' :: f \to m' :: f \quad (\texttt{rej}) \tag{C}$$

We let m range over EKW machine configurations (with $\mathring{m}$ representing the initial EKW configuration), the labels within brackets indicate the possible EKWF transitions matching the final step. For all three forms, the inductive hypothesis yields the following EKW run:

$$\mathring{m} \xrightarrow{*} m$$

For cases (A) and (B), the involved EKWF transitions (in the final step) have corresponding (non-deterministic) counterparts in the EKW machine (Figure 3.2) of the form $m \rightarrow m'$, which immediately leads to the desired result. For case (C), we argue that the EKW configuration m' uncovered in the n^{th} step must have been introduced to the EKWF run at some earlier step. If we let this be the k^{th} step ($k < n$), we have a run of the following form:

$$\mathring{m} :: [\,] \xrightarrow{k-1} \bar{m} :: f' \rightarrow m'' :: m' :: f' \xrightarrow{n-k} m' :: f$$

Here we note that the k^{th} EKWF transition must be one of `alt` or `kln`, for which the corresponding (non-deterministic) EKW transitions imply $\bar{m} \rightarrow m'$. Since the induction hypothesis applied to the first $k-1$ steps yields $\mathring{m} \xrightarrow{*} \bar{m}$, the result holds for (C) as well. □

Lemma 3.2.10 (EKWF partial correctness) If there is an EKWF run:

$$\langle e \;;\; [\,] \;;\; w \rangle :: [\,] \xrightarrow{*} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle :: f$$

Then $e \downarrow w$.

Proof Applying Lemma 3.2.9 to the given EKWF run gives the following EKW run:

$$\langle e \;;\; [\,] \;;\; w \rangle \xrightarrow{*} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

Applying Lemma 3.2.6 to this run establishes the required result. □

3.2.1 Termination

There is no guarantee that either of the machines will terminate on all inputs, which is a requirement for total correctness. In fact, there are valid inputs

$$\langle e_1 \mid e_2 \,;\, k \,;\, w\rangle :: f \xrightarrow{\texttt{alt}} \langle e_1 \,;\, k \,;\, w\rangle :: \langle e_2 \,;\, k \,;\, w\rangle :: f$$

$$\langle e_1\, e_2 \,;\, k \,;\, w\rangle :: f \xrightarrow{\texttt{conc}} \langle e_1 \,;\, e_2 :: k \,;\, w\rangle :: f$$

$$\langle e^* \,;\, k \,;\, w\rangle :: f \xrightarrow{\texttt{kln}} \langle \varepsilon \,;\, k \,;\, w\rangle :: \langle e \,;\, e^* :: k \,;\, w\rangle :: f$$

$$\langle a \,;\, k \,;\, a\, w\rangle :: f \xrightarrow{\texttt{match}} \langle \varepsilon \,;\, k \,;\, w\rangle :: f$$

$$\langle \varepsilon \,;\, e :: k \,;\, w\rangle :: f \xrightarrow{\texttt{pop}} \langle e \,;\, k \,;\, w\rangle :: f$$

$$m :: f \xrightarrow{\texttt{rej}} f \quad \text{where}$$
$$m = \langle a \,;\, k \,;\, b\, w\rangle \text{ or}$$
$$m = \langle a \,;\, k \,;\, \varepsilon\rangle \text{ or}$$
$$m = \langle \varepsilon \,;\, [\,] \,;\, a\, w\rangle$$

Figure 3.3: EKWF transitions

on which the EKW machine *may* enter an infinite loop; a trivial example is the configuration $\langle a^{**} \,;\, [\,] \,;\, a\rangle$:

$$\langle a^{**} \,;\, [\,] \,;\, a\rangle \xrightarrow{\texttt{kln1}} \langle a^{*} \,;\, a^{**} \,;\, a\rangle \xrightarrow{\texttt{kln2}} \langle \varepsilon \,;\, a^{**} \,;\, a\rangle \xrightarrow{\texttt{pop}} \langle a^{**} \,;\, [\,] \,;\, a\rangle$$

This problem is further exacerbated when it comes to the EKWF machine; while the EKW machine *may* enter an infinite loop for the said input, the EKWF machine *always* enters an infinite loop for the same input due to the elimination of non-determinism:

$$\begin{aligned}
\langle a^{**} \,;\, [\,] \,;\, a\rangle :: f &\xrightarrow{\texttt{kln}} \langle \varepsilon \,;\, [\,] \,;\, a\rangle :: \langle a^{*} \,;\, a^{**} \,;\, a\rangle :: f \\
&\xrightarrow{\texttt{rej}} \langle a^{*} \,;\, a^{**} \,;\, a\rangle :: f \\
&\xrightarrow{\texttt{kln}} \langle \varepsilon \,;\, a^{**} \,;\, a\rangle :: \langle a \,;\, a^{*} :: a^{**} \,;\, a\rangle :: f \\
&\xrightarrow{\texttt{pop}} \langle a^{**} \,;\, [\,] \,;\, a\rangle :: \langle a \,;\, a^{*} :: a^{**} \,;\, a\rangle :: f \\
&\longrightarrow \ldots
\end{aligned}$$

The importance of this issue is witnessed by the fact that even the most popular regex libraries such as PCRE [Haz12a] has been battered by similar "infinite recursion" problems from time to time. According to the PCRE change log [Haz12b], variants of this infinite looping problem has been "fixed" in several occasions. We believe that the core of this problem is reflected in the example highlighted above, and we intend to provide a systematic fix that ensures the correct termination of both the EKW and EKWF machines.

We begin by observing that all infinite loops are caused by the following EKW transition sequence:

$$\langle e^* \,;\, k \,;\, w\rangle \xrightarrow{\texttt{kln1}} \langle e \,;\, e^* :: k \,;\, w\rangle \xrightarrow{*} \langle \varepsilon \,;\, e^* :: k \,;\, w\rangle \xrightarrow{\texttt{pop}} \langle e^* \,;\, k \,;\, w\rangle \rightarrow \ldots$$

Intuitively, an infinite loop occurs whenever the inner expression e of a Kleene expression e^* is nullable, i.e., e matches the empty string. Our proposed solution introduces a *barrier expression* into the problematic Kleene transition as illustrated below:

$$\langle e^* \,;\, k \,;\, w\rangle \xrightarrow{\texttt{kln2}'} \langle e \,;\, \lambda^i :: e^* :: k \,;\, w\rangle \quad \{i = |w|\}$$

The behavior of the barrier expression is given by the following transition:

$$\langle \lambda^i \,;\, e :: k \,;\, w\rangle \xrightarrow{\texttt{barr}} \langle e \,;\, k \,;\, w\rangle \quad \text{if } \{i > |w|\}$$

Intuitively, the barrier ensures that the inner expression does not match the empty string. For example, it successfully terminates the infinite loop (rejects the run) caused by the $\langle a^{**} \,;\, [\,] \,;\, a\rangle$ configuration as shown below:

$$\begin{aligned}
\langle a^{**} \,;\, [\,] \,;\, a\rangle &\xrightarrow{\texttt{kln1}} \langle a^* \,;\, \lambda^1 :: a^{**} \,;\, a\rangle \\
&\xrightarrow{\texttt{kln2}} \langle \varepsilon \,;\, \lambda^1 :: a^{**} \,;\, a\rangle \\
&\xrightarrow{\texttt{pop}} \langle \lambda^1 \,;\, a^{**} \,;\, a\rangle \qquad \text{(reject)}
\end{aligned}$$

Definition 3.2.11 The abstract syntax of regular expressions in Definition 1.1.1 is extended to include barrier expressions as follows:

$$\mathcal{L}(\lambda^i) = \{\varepsilon\} \qquad i > |w|$$

Where w is the remainder of the input string to be matched. Therefore, λ^i has the big-step semantics:

$$\frac{i > |w|}{\lambda^i \downarrow \varepsilon}$$

It may be helpful to think of λ^i as an internal expression used solely for the operation of the backtracking matcher and not something exposed to the users. Whether this use of λ^i within the backtracking matcher has any impact on the normal operation of the matcher (other than eliminating nontermination) needs to be considered separately (Section 7.3).

If we try to define a termination measure on the basic EKW machine (i.e without barriers), we run into a circularity, since the Kleene star can cause e^* to be pushed and popped from the stack forever. It is exactly this behavior that gives rise to the nontermination. If this form of looping is prevented, we are able to define a termination measure by taking into account where the loop is broken. This is where the barrier expressions come into the play.

For each machine state, we compute an upper bound on the number of steps the machine can take before consuming input. This bound can be statically computed from e and k. We have to verify that this number strictly decreases in any step of a run as long as w stays the same. Then a lexicographic order on the length of w and this bound gives us termination. The bound may go up sometimes, but only when w decreases.

Definition 3.2.12 We define a function $\mathbf{S}(e, k, w)$ recursive on e and k as

follows:

$$\mathbf{S}(a, k, w) = 0 \tag{3.2.12.1}$$

$$\mathbf{S}(\varepsilon, [\,], w) = 0 \tag{3.2.12.2}$$

$$\mathbf{S}(\varepsilon, e :: k, w) = \mathbf{S}(e, k, w) + 1 \tag{3.2.12.3}$$

$$\mathbf{S}(\lambda^i, k, w) = 0 \qquad i \leq |w| \text{ or } k = [\,] \tag{3.2.12.4}$$

$$\mathbf{S}(\lambda^i, e :: k, w) = \mathbf{S}(e, k, w) + 1 \qquad i > |w| \tag{3.2.12.5}$$

$$\mathbf{S}(e^*, k, w) = \max(\mathbf{S}(\varepsilon, k, w), \mathbf{S}(e, [\,], w) + 1) + 1 \tag{3.2.12.6}$$

$$\mathbf{S}((e_1 \mid e_2), k, w) = \max(\mathbf{S}(e_1, k, w), \mathbf{S}(e_2, k, w)) + 1 \tag{3.2.12.7}$$

$$\mathbf{S}((e_1\, e_2), k, w) = \mathbf{S}(e_1, e_2 :: k, w) + 1 \tag{3.2.12.8}$$

The barrier lets us ignore the step measure of the continuation in rule 3.2.12.6, which avoids the circularity for Kleene transitions. Note that in the same rule, the $(+1)$ in $\mathbf{S}(e, [\,], w)+1$ arises from the extra step required to pop the barrier.

Lemma 3.2.13 The recursive function $\mathbf{S}(e, k, w)$ is well defined.

Proof First, observe that Definition 3.2.12 have rules for all possible (e, k) combinations; most rules are recursive on e alone and for those depending on a non-empty k, a separate rule covers the empty k case.

Secondly, in each rule of Definition 3.2.12, the number of syntax tree nodes in RHS $S(\,)$ invocations is strictly less than that on the LHS. In rule 3.2.12.3 the syntax tree node for ε is missing in the RHS, in rule 3.2.12.8 the node for concatenation is missing in the RHS. The remaining rules follow similarly. □

Lemma 3.2.14 For all regular expressions e, continuations k, k' and input

strings w such that $i = |w|$,

$$n = \mathbf{S}(e, k :: \lambda^i :: k', w) \Longrightarrow n \leq \mathbf{S}(e, k, w) + 1$$

Proof We peform an induction on the value n. The applicable base cases ($n = 0$) satisfy the result as shown below:

$$\begin{aligned}
n &= \mathbf{S}(a, k :: \lambda^i :: k', w) \\
&= 0 \\
&\leq 1 \\
&\leq \mathbf{S}(a, k, w) + 1 \\
n &= \mathbf{S}(\lambda^j, k :: \lambda^i :: k', w) \quad \{j = |w|\} \\
&= 0 \\
&\leq 1 \\
&\leq \mathbf{S}(\lambda^j, k, w) + 1 \quad \{j = |w|\}
\end{aligned}$$

For the inductive step, we perform a case analysis of n on the L.H.S:

- Case 3.2.12.3:

$$\begin{aligned}
n &= \mathbf{S}(\varepsilon, e :: k :: \lambda^i :: k', w) \\
&= \mathbf{S}(e, k :: \lambda^i :: k', w) + 1 \quad \{\text{Def. 3.2.12.3}\} \\
&\leq \mathbf{S}(e, k, w) + 1 + 1 \quad \{\text{I.H}\} \\
&\leq \mathbf{S}(\varepsilon, e :: k, w) + 1 \quad \{\text{Def. 3.2.12.3}\}
\end{aligned}$$

- Case 3.2.12.5:

$$
\begin{aligned}
n &= \mathbf{S}(\lambda^j, e :: k :: \lambda^i :: k', w) \quad \{j > |w|\} \\
&= \mathbf{S}(e, k :: \lambda^i :: k', w) + 1 \quad \{\text{Def. 3.2.12.5}\} \\
&\leq \mathbf{S}(e, k, w) + 1 + 1 \quad \{\text{I.H}\} \\
&\leq \mathbf{S}(\lambda^j, e :: k, w) + 1 \quad \{\text{Def. 3.2.12.5}\}
\end{aligned}
$$

- Case 3.2.12.6:

$$
\begin{aligned}
n &= \mathbf{S}(e^*, k :: \lambda^i :: k', w) \\
&= \max(\mathbf{S}(e, [\,], w) + 1, \mathbf{S}(\varepsilon, k :: \lambda^i :: k', w)) + 1 \quad \{\text{Def. 3.2.12.6}\} \\
&\leq \max(\mathbf{S}(e, [\,], w) + 1, \mathbf{S}(\varepsilon, k, w) + 1) + 1 \quad \{\text{I.H}\} \\
&\leq \max(\mathbf{S}(e, [\,], w), \mathbf{S}(\varepsilon, k, w)) + 1 + 1 \\
&\leq \max(\mathbf{S}(e, [\,], w) + 1, \mathbf{S}(\varepsilon, k, w)) + 1 + 1 \\
&\leq \mathbf{S}(e^*, k, w) + 1 \quad \{\text{Def. 3.2.12.6}\}
\end{aligned}
$$

- Case 3.2.12.7:

$$
\begin{aligned}
n &= \mathbf{S}((e_1 \mid e_2), k :: \lambda^i :: k', w) \\
&= \max(\mathbf{S}(e_1, k :: \lambda^i :: k', w), \mathbf{S}(e_2, k :: \lambda^i :: k', w)) + 1 \quad \{\text{Def. 3.2.12.7}\} \\
&\leq \max(\mathbf{S}(e_1, k, w) + 1, \mathbf{S}(e_2, k, w) + 1) + 1 \quad \{\text{I.H}\} \\
&\leq \max(\mathbf{S}(e_1, k, w), \mathbf{S}(e_2, k, w)) + 1 + 1 \\
&\leq \mathbf{S}((e_1 \mid e_2), k, w) + 1 \quad \{\text{Def. 3.2.12.7}\}
\end{aligned}
$$

- Case 3.2.12.8:

$$\begin{aligned} n &= \mathbf{S}((e_1\, e_2), k :: \lambda^i :: k', w) \\ &= \mathbf{S}(e_1, e_2 :: k :: \lambda^i :: k', w) + 1 \quad \{\text{Def. 3.2.12.8}\} \\ &\leq \mathbf{S}(e_1, e_2 :: k, w) + 1 + 1 \quad \{\text{I.H}\} \\ &\leq \mathbf{S}(e_1\, e_2, k, w) + 1 \quad \{\text{Def. 3.2.12.8}\} \end{aligned}$$

□

Lemma 3.2.15 The quantity $\mathbf{S}(e, k, w)$ strictly decreases for any silent (non-character consuming) transition of the EKW machine.

Proof This holds true for most of the silent EKW transitions by the corresponding S-measure in Definition 3.2.12. The only exception to this rule is the (augmented) Kleene transition:

$$\langle e^* \,;\, k \,;\, w\rangle \xrightarrow{\texttt{kln2}'} \langle e \,;\, \lambda^i :: k \,;\, w\rangle \quad \{i = |w|\}$$

In this case, we apply Lemma 3.2.14 to the R.H.S:

$$\begin{aligned} S_{R.H.S} &= \mathbf{S}(e, \lambda^i :: k, w) \\ &\leq \mathbf{S}(e, [\,], w) + 1 \\ &< \mathbf{S}(e^*, k, w) \quad \{\text{Definition 3.2.12.6}\} \end{aligned}$$

□

Theorem 3.2.16 Any EKW machine configuration $\langle e \,;\, [\,] \,;\, w\rangle$ (with barrier expressions) always terminates.

Proof The EKW machine with barrier expressions has only two kinds of transitions: those consuming input and input-free transitions that do not

$$\frac{t \in T \qquad m_0 \xrightarrow{t} m \qquad m \xrightarrow{\texttt{alt1}} m_1 \qquad m \xrightarrow{\texttt{alt2}} m_2}{T \xrightarrow{[\texttt{alt}]} \{(t :: [\texttt{alt1}]), (t :: [\texttt{alt2}])\} \cup T} \qquad \frac{T_1 \xrightarrow{t_1} T_2 \qquad T_2 \xrightarrow{t_2} T_3}{T_1 \xrightarrow{t_1 :: t_2} T_3}$$

$$\frac{t \in T \qquad m_0 \xrightarrow{t} m \qquad m \xrightarrow{\texttt{kln1}} m_1 \qquad m \xrightarrow{\texttt{kln2}'} m_2}{T \xrightarrow{[\texttt{kln}]} \{(t :: [\texttt{kln1}]), (t :: [\texttt{kln2}'])\} \cup T}$$

$$\frac{t \in T \qquad m_0 \xrightarrow{t} m \qquad m \xrightarrow{r} m_1 \qquad r \in \{\texttt{conc}, \texttt{match}, \texttt{barr}, \texttt{pop}\}}{T \xrightarrow{[r]} \{(t :: [r])\} \cup T}$$

$$\frac{}{T \xrightarrow{[\texttt{rej}]} T} \qquad \frac{}{T \xrightarrow{[\,]} T}$$

Figure 3.4: EKWF Traces

affect the input. By extension, any EKW machine run consists of input-free runs interspersed with character-consuming transitions. From Lemma 3.2.15 it follows that all input-free runs consist of finitely many input-free transitions. Since w is also finite, the number of such input-free runs and the number of input-consuming transitions must also be finite. Hence, all runs terminate. □

Definition 3.2.17 (EKW traces) An EKW trace is a sequence of transition labels of the form:

$$m_0 \xrightarrow{[r_1, r_2, \ldots, r_k]} m_k$$

Traces are defined inductively using the following rules:

$$\frac{m_1 \xrightarrow{r} m_2}{m_1 \xrightarrow{[r]} m_2} \qquad \frac{}{m \xrightarrow{[\,]} m} \qquad \frac{m_1 \xrightarrow{t_1} m_2 \qquad m_2 \xrightarrow{t_2} m_3}{m_1 \xrightarrow{t_1 :: t_2} m_3}$$

All the EKW rules are deterministic for a given trace. For a given initial configuration m_0 and a trace t, there is a unique m such that $m_0 \xrightarrow{t} m$.

Definition 3.2.18 (EKW Trees) Let m_0 be an EKW configuration. We

define the EKW tree of m_0 as:

$$\downarrow m_0 \stackrel{\text{def}}{=} \{t \mid \exists m.m_0 \xrightarrow{t} m\}$$

The definition of $\downarrow m_0$ is much like the usual notion of downset. However, we do not form this set by collecting the reachable configurations m. Instead, we record the trace that leads to m (from which m may be recovered). This definition gives us more structure, which will be useful in the EKWF termination proof.

Definition 3.2.19 (EKWF traces) An EKWF trace is of the form $T_1 \xrightarrow{q} T_2$, where q is a sequence of EKWF transition labels, and both T_1 and T_2 are sets of EKW traces. EKWF traces are defined by the rules given in Figure 3.4.

Intuitively, the EKWF machine computes EKW trees by exploring all alternatives. The definition of EKWF traces captures how each move of the EKWF machine expands this search tree.

Theorem 3.2.20 (EKWF termination) For each configuration of the EKWF machine reachable from $m_0 :: [\,]$ via an EKWF trace q with $\{[\,]\} \xrightarrow{q} T$, the lexicographically ordered pair of integers:

$$((\#\downarrow m_0 - \# T), |f|)$$

strictly decreases, where $|f|$ is the size of the EKWF machine (i.e. the number of EKW frames in it).

Proof The proof follows from two key observations:

- $\#\downarrow m_0$ should be finite. For if it is not, it would mean $\downarrow m_0$ corresponds

to a finitely branching infinite tree, which according to König's Lemma should contain an infinite EKW trace. This forms a contradiction given EKW termination (Theorem 3.2.16).

- $\#T$ increases for all the EKWF transitions except `rej`, for which it remains constant. But for those transitions $|f|$ decreases.

□

3.2.2 Exponential runtime

We take the example of matching regular expression a^{**} against the input string $a^n b$ and show that the backtracking EKWF machine results in an exponential number of transitions with respect to n.

Definition 3.2.21 For natural numbers u, v and n, let $c_{u,v,n}$ be the EKW configuration:

$$\langle a \ ; \ \lambda^u :: a^* :: \lambda^v :: a^{**} \ ; \ a^n b \rangle$$

Lemma 3.2.22 If $u, v > n > 0$, then

$$c_{u,v,n} :: f \rightarrow^{13} c_{n,n,n-1} :: c_{n,v,n-1} :: f$$

Proof By simulating the initial configuration for 13 steps:

$$
\begin{aligned}
& \langle a \;;\; \lambda^u :: a^* :: \lambda^v :: a^{**} \;;\; a^n b \rangle :: f \\
\xrightarrow{\texttt{match}} \quad & \langle \varepsilon \;;\; \lambda^u :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{pop}} \quad & \langle \lambda^u \;;\; a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{barr}} \quad & \langle \varepsilon \;;\; a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{pop}} \quad & \langle a^* \;;\; \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{kln}} \quad & \langle \varepsilon \;;\; \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{pop}} \quad & \langle \lambda^v \;;\; a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{barr}} \quad & \langle \varepsilon \;;\; a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{pop}} \quad & \langle a^{**} \;;\; [\,] \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{kln}} \quad & \langle \varepsilon \;;\; [\,] \;;\; a^{n-1} b \rangle :: \langle a^* \;;\; \lambda^n :: a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{rej}} \quad & \langle a^* \;;\; \lambda^n :: a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{kln}} \quad & \langle \varepsilon \;;\; \lambda^n :: a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^n :: a^{**} \;;\; a^{n-1} b \rangle :: \\
& \qquad \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{pop}} \quad & \langle \lambda^n \;;\; a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^n :: a^{**} \;;\; a^{n-1} b \rangle :: \\
& \qquad \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f \\
\xrightarrow{\texttt{rej}} \quad & \langle a \;;\; \lambda^n :: a^* :: \lambda^n :: a^{**} \;;\; a^{n-1} b \rangle :: \langle a \;;\; \lambda^n :: a^* :: \lambda^v :: a^{**} \;;\; a^{n-1} b \rangle :: f
\end{aligned}
$$

□

Lemma 3.2.23 For $u, v > n \geq 0$, the EKWF machine $c_{u,v,n} :: f$ must transition to f, and takes at least 2^n steps to do so. More formally, there exists q such that $q \geq 2^n$ and:

$$c_{u,v,n} :: f \to^q f$$

Since the EKWF machine is deterministic, this statement gives us a lower

bound on its runtime.

Proof Follows from Lemma 3.2.22. The top-most EKW frame splits into two equally sized problems after each 13 steps while n only keeps decreasing by one. □

Example 3.2.24 The EKWF configuration $\langle a^{**} \,;\, [\,] \,;\, a^n b\rangle :: [\,]$ takes an exponential number of transitions for termination with respect to n.

Consider the partial run:

$$
\begin{aligned}
&\langle a^{**} \,;\, [\,] \,;\, a^n b\rangle \\
&\quad \xrightarrow{\texttt{kln}} \langle \varepsilon \,;\, [\,] \,;\, a^n b\rangle :: \langle a^{*} \,;\, \lambda^{(n+1)} :: a^{**} \,;\, a^n b\rangle \\
&\quad \xrightarrow{\texttt{rej}} \langle a^{*} \,;\, \lambda^{(n+1)} :: a^{**} \,;\, a^n b\rangle \\
&\quad \xrightarrow{\texttt{kln}} \langle \varepsilon \,;\, \lambda^{(n+1)} :: a^{**} \,;\, a^n b\rangle :: \langle a \,;\, \lambda^{(n+1)} :: a^{*} :: \lambda^{(n+1)} :: a^{**} \,;\, a^n b\rangle \\
&\quad \xrightarrow{\texttt{pop}} \langle \lambda^{(n+1)} \,;\, a^{**} \,;\, a^n b\rangle :: \langle a \,;\, \lambda^{(n+1)} :: a^{*} :: \lambda^{n+1} :: a^{**} \,;\, a^n b\rangle \\
&\quad \xrightarrow{\texttt{rej}} \langle a \,;\, \lambda^{(n+1)} :: a^{*} :: \lambda^{(n+1)} :: a^{**} \,;\, a^n b\rangle
\end{aligned}
$$

At this point, applying the result of Lemma 3.2.23 yields the desired result.

3.3 The lockstep machine

We refine the EKW machine by representing the regular expression as a data structure in a heap π, which serves as the program run by the machine. That way, the machine can distinguish between different positions in the syntax tree.

Definition 3.3.1 A heap π is a finite partial function from addresses to

p	$\pi(p)$	$\texttt{cont}\, p$
p_0	$p_1 \bullet p_2$	`null`
p_1	${p_3}^*$	p_2
p_2	b	`null`
p_3	${p_4}^*$	p_1
p_4	a	p_3

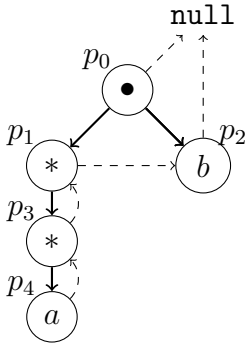


Figure 3.5: The regular expression $a^{**} \bullet b$ as a tree with continuation pointers

values. There exists a distinguished address `null`, which is not mapped to any value.

In our setting, the values are syntax tree nodes, represented by an operator from the syntax of regular expressions together with pointers to the tree for the arguments (if any) of the operator. For example, for sequential composition, we have a node containing $(p_1 \bullet p_2)$, where the two pointers p_1 and p_2 point to the trees of the two expressions being composed.

Definition 3.3.2 We write $\otimes$ for the partial operation of forming the union of two partial functions provided that their domains are disjoint. More formally, let $f_1 : A \rightharpoonup B$ and $f_2 : A \rightharpoonup B$ be two partial functions. Then if $\mathrm{dom}(f_1) \cap \mathrm{dom}(f_2) = \emptyset$, the function

$$(f_1 \otimes f_2) : A \rightharpoonup B$$

is defined as $f_1 \otimes f_2 = f_1 \cup f_2$.

Note that $\otimes$ is the same as the operation $*$ on heaps in separation logic [Rey02], and hence a commutative partial monoid. We avoid the notation $*$ as it could be confused with the Kleene star. As in separation logic,

we use $\otimes$ to describe data structures with pointers in memory.

Definition 3.3.3 We write $\pi, p \models e$ if p points to the root node of a regular expression e in a heap π. The relation is defined by induction on e as follows:

$$
\begin{array}{lll}
\pi, p & \models a & \text{if } \pi(p) = a \\
\pi, p & \models \varepsilon & \text{if } \pi(p) = \varepsilon \\
\pi, p & \models (e_1 \mid e_2) & \text{if } \pi = \pi_0 \otimes \pi_1 \otimes \pi_2 \wedge \pi_0(p) = (p_1 \mid p_2) \\
 & & \wedge\, \pi_1, p_1 \models e_1 \wedge \pi_2, p_2 \models e_2 \\
\pi, p & \models (e_1\, e_2) & \text{if } \pi = \pi_0 \otimes \pi_1 \otimes \pi_2 \wedge \pi_0(p) = (p_1 \bullet p_2) \\
 & & \wedge\, \pi_1, p_1 \models e_1 \wedge \pi_2, p_2 \models e_2 \\
\pi, p & \models {e_1}^* & \text{if } \pi = \pi_0 \otimes \pi_1 \wedge \pi_0(p) = {p_1}^* \wedge \pi_1, p_1 \models e_1
\end{array}
$$

Here the definition of $\pi, p \models e$ precludes any cycles in the child pointer chain.

As an example, consider the regular expression $e = a^{**}b$. A π and p_0 such that $\pi, p_0 \models e$ is given by the table in Figure 3.5. The tree structure, represented by the solid arrows, is drawn on the right.

Definition 3.3.4 Let `cont` be a function:

$$\texttt{cont} : \mathrm{dom}(\pi) \to (\mathrm{dom}(\pi) \cup \{\texttt{null}\})$$

Where,

$$\texttt{cont}\, p' = \begin{cases} \texttt{null} & \text{if } p' \text{ is root} \\ \texttt{cont}\, p & \text{if } \exists p \in \mathrm{dom}(\pi)\,.\,\pi(p) = (p' \mid p'') \\ \texttt{cont}\, p & \text{if } \exists p \in \mathrm{dom}(\pi)\,.\,\pi(p) = (p'' \mid p') \\ p'' & \text{if } \exists p \in \mathrm{dom}(\pi)\,.\,\pi(p) = (p' \bullet p'') \\ \texttt{cont}\, p & \text{if } \exists p \in \mathrm{dom}(\pi)\,.\,\pi(p) = (p'' \bullet p') \\ \texttt{cont}\, p & \text{if } \exists p \in \mathrm{dom}(\pi)\,.\,\pi(p) = p'^{*} \end{cases}$$

Lemma 3.3.5 The function `cont` is well defined.

Proof A node p' can either be on its own (in which case it is the root of the tree) or be a sub-expression of a larger parent expression (Alternation, Concatenation or Kleene). Observe that Definition 3.3.4 contains a case statement for each of these possiblities. Secondly, each recursive `cont` invocation on the RHS refers to the parent node of the current node. Given that Definition 3.3.3 precludes any cycles in the parent-child pointer chain, all such calls terminate. □

The function `cont` can be easily computed by a recursive tree walk. We elide it when it is clear from the context, assuming that π always comes equipped with a `cont`. By treating `cont` as a function, we have not committed to a particular implementation; for instance `cont` could be represented as a hash table indexed by pointer values, or it could be added as another pointer field to the nodes in the heap.

In the example presented in Figure 3.5, dashed arrows represent `cont`. In particular, note the cycle leading downward from p_1 and up again via dashed

$\boxed{p \longrightarrow q \text{ or } p \xrightarrow{a} q \text{ relative to } \pi}$

$$
\begin{array}{lcll}
p & \longrightarrow & p_1 & \text{if } \pi(p) = p_1 \mid p_2 \\
p & \longrightarrow & p_2 & \text{if } \pi(p) = p_1 \mid p_2 \\
p & \longrightarrow & p_1 & \text{if } \pi(p) = p_1 \bullet p_2 \\
p & \longrightarrow & p_1 & \text{if } \pi(p) = {p_1}^* \\
p & \longrightarrow & p_2 & \text{if } \pi(p) = {p_1}^* \text{ and } \texttt{cont}\, p = p_2 \\
p & \longrightarrow & p_1 & \text{if } \pi(p) = \varepsilon \text{ and } \texttt{cont}\, p = p_1 \\
p & \xrightarrow{a} & p' & \text{if } \pi(p) = a \text{ and } \texttt{cont}\, p = p'
\end{array}
$$

Figure 3.6: PWπ transitions

arrows. Following such a cycle could lead to infinite loops as with the EKW machine presented earlier.

Definition 3.3.6 The PWπ machine is defined as follows. Transitions of this machine are always relative to some heap π, which does not change during evaluation. We elide π if it is clear from the context. Configurations of the machine are of the form $\langle p \ ; \ w \rangle$, where p is a pointer in π and w is a string of input symbols. Given the transition relation between pointers defined in Figure 3.6, the machine has the following transitions:

$$\frac{p \xrightarrow{a} q}{\langle p \ ; \ a \ w \rangle \rightarrow \langle q \ ; \ w \rangle} \qquad \frac{p \longrightarrow q}{\langle p \ ; \ w \ \rangle \rightarrow \langle q \ ; \ w \rangle}$$

The accepting state of the machine is $\langle \texttt{null} \ ; \ \varepsilon \rangle$. That is, both the continuation and the remaining input have been consumed.

Intuitively, the transitions in Figure 3.6 model the path that can be taken by a regular expression matcher at each node. Instead of pushing and poping a continuation stack (K) as in the EKW machine, the PWπ machine encodes them into a tree structure (π).

Example 3.3.7 For a regular expression $e = a^{**}b$, let π and p_0 be such that $\pi, p_0 \models e$. See Figure 3.5 for the representation of π as a tree with pointers. The diagram below illustrates two possible executions of the PWπ machine against inputs e and aab.

Execution - 1: Infinite loop	Execution - 2: Successful match
$\langle p_0 \,;\, aab \rangle$	$\langle p_0 \,;\, aab \rangle$
$\longrightarrow \langle p_1 \,;\, aab \rangle$	$\longrightarrow \langle p_1 \,;\, aab \rangle$
$\longrightarrow \langle p_3 \,;\, aab \rangle$	$\longrightarrow \langle p_3 \,;\, aab \rangle$
$\longrightarrow \langle p_1 \,;\, aab \rangle$	$\longrightarrow \langle p_4 \,;\, aab \rangle$
$\longrightarrow \langle p_3 \,;\, aab \rangle$	$\longrightarrow \langle p_3 \,;\, ab \rangle$
$\longrightarrow \langle p_1 \,;\, aab \rangle$	$\longrightarrow \langle p_4 \,;\, ab \rangle$
$\longrightarrow \langle p_3 \,;\, aab \rangle$	$\longrightarrow \langle p_3 \,;\, b \rangle$
$\longrightarrow \langle p_1 \,;\, aab \rangle$	$\longrightarrow \langle p_1 \,;\, b \rangle$
$\longrightarrow \langle p_3 \,;\, aab \rangle$	$\longrightarrow \langle p_2 \,;\, b \rangle$
$\longrightarrow \langle p_1 \,;\, aab \rangle$	$\longrightarrow \langle \texttt{null} \,;\, \varepsilon \rangle$
$\longrightarrow \quad \ldots$	

Definition 3.3.8 For a pointer a p, we define the function $\texttt{stack}\, p$ as follows:

$$\begin{array}{lcll} \texttt{stack}\, p & = & [\,] & \text{if } \texttt{cont}\, p = \texttt{null} \\ \texttt{stack}\, p & = & e :: (\texttt{stack}\, q) & \text{if } q = \texttt{cont}\, p \neq \texttt{null} \\ & & & \text{and } \pi, q \models e \end{array}$$

Lemma 3.3.9 Let π be a heap such that $\pi, p \models e$ and $\texttt{stack}\, p = k$. Then

there is a run of the EKW machine of the form

$$\langle e \;;\; k \;;\; w \rangle \xrightarrow{*} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

if and only if there is a run of the $PW\pi$ machine of the form

$$\langle p \;;\; w \rangle \xrightarrow{*} \langle \texttt{null} \;;\; \varepsilon \rangle$$

Proof By induction over the length of the runs. For the forward implication, suppose:

$$\langle e \;;\; k \;;\; w \rangle \xrightarrow{n} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

The applicable base cases ($n = 1$) are as follows:

$$\langle a \;;\; [\,] \;;\; a \rangle \xrightarrow{\texttt{match}} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

$$\langle (\varepsilon \mid e) \;;\; [\,] \;;\; \varepsilon \rangle \xrightarrow{\texttt{alt1}} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

$$\langle (e \mid \varepsilon) \;;\; [\,] \;;\; \varepsilon \rangle \xrightarrow{\texttt{alt2}} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

$$\langle e^* \;;\; [\,] \;;\; \varepsilon \rangle \xrightarrow{\texttt{kln1}} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

$$\langle \varepsilon \;;\; [\varepsilon] \;;\; \varepsilon \rangle \xrightarrow{\texttt{pop}} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

In each of these cases, it is quite straightforward to construct a $PW\pi$ machine run of the required form (details omitted to avoid clutter). For the inductive step, we perform a case analysis of e:

- $e = \varepsilon$:

  $$\langle \varepsilon \;;\; e' :: k' \;;\; w \rangle \xrightarrow{n} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

  Where $\pi, p \models \varepsilon$ and $\texttt{stack}\, p = e' :: k'$. Taking $\texttt{cont}\, p = q$, the definition of $\texttt{stack}()$ gives $\pi, q \models e'$ and $\texttt{stack}\, q = k'$. Furthermore, from the EKW machine (Figure 3.2) we get:

  $$\langle \varepsilon \;;\; e' :: k' \;;\; w \rangle \xrightarrow{\texttt{pop}} \langle e' \;;\; k' \;;\; w \rangle \xrightarrow{n-1} \langle \varepsilon \;;\; [\,] \;;\; \varepsilon \rangle$$

Therefore, from the $PW\pi$ transitions (Figure 3.6) and the induction hypothesis we get:

$$\langle p \; ; \; w \rangle \rightarrow \langle q \; ; \; w \rangle \xrightarrow{*} \langle \texttt{null} \; ; \; \varepsilon \rangle$$

- $e = a$:

$$\langle a \; ; \; e' :: k' \; ; \; aw' \rangle \xrightarrow{n} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Where $\pi, p \models a$ and $\texttt{stack}\, p = e' :: k'$. As before, taking $\texttt{cont}\, p = q$, we get $\pi, q \models e'$ and $\texttt{stack}\, q = k'$. Then from the EKW machine we get:

$$\langle a \; ; \; e' :: k' \; ; \; aw' \rangle \xrightarrow{\texttt{match}} \langle \varepsilon \; ; \; e' :: k' \; ; \; w \rangle \xrightarrow{\texttt{pop}} \langle e' \; ; \; k' \; ; \; w \rangle \xrightarrow{n-2} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Again from the $PW\pi$ transitions and the induction hypothesis we derive:

$$\langle p \; ; \; aw \rangle \rightarrow \langle q \; ; \; w \rangle \xrightarrow{*} \langle \texttt{null} \; ; \; \varepsilon \rangle$$

- $e = (e_1 \mid e_2)$:

$$\langle (e_1 \mid e_2) \; ; \; k \; ; \; w \rangle \xrightarrow{n} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Where $\pi, p \models (e_1 \mid e_2)$ and $\texttt{stack}\, p = k$. Suppose:

$$\pi, p_1 \models e_1 \quad \pi, p_2 \models e_2$$

So that: $\pi(p) = (p_1 \mid p_2)$. Then from the definition of $\texttt{cont}()$ we get:

$$\texttt{cont}\, p_1 = \texttt{cont}\, p_2 = \texttt{cont}\, p$$

Which in turn suggests:

$$\texttt{stack}\, p_1 = \texttt{stack}\, p_2 = k$$

Without loss of generality, from the EKW transitions:

$$\langle (e_1 \mid e_2) \; ; \; k \; ; \; w \rangle \xrightarrow{\texttt{alt1}} \langle e_1 \; ; \; k \; ; \; w \rangle \xrightarrow{n-1} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Now from the $PW\pi$ transitions and the inductive hypothesis we get:

$$\langle p \; ; \; w \rangle \rightarrow \langle p_1 \; ; \; w \rangle \xrightarrow{*} \langle \texttt{null} \; ; \; \varepsilon \rangle$$

- $e = e_1 e_2$:

$$\langle e_1 e_2 \; ; \; k \; ; \; w \rangle \xrightarrow{n} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Where $\pi, p \models e_1 e_2$ and $\texttt{stack}\, p = k$. Suppose:

$$\pi, p_1 \models e_1 \quad \pi, p_2 \models e_2$$

So that: $\pi(p) = p_1 \bullet p_2$. Then from the defintion of $\texttt{cont}()$ we get:

$$\texttt{cont}\, p_1 = p_2 \quad \texttt{cont}\, p_2 = \texttt{cont}\, p$$

Which in turn suggests: $\texttt{stack}\, p_1 = e_2 :: k$. Now from the EKW machine transitions:

$$\langle e_1 e_2 \; ; \; k \; ; \; w \rangle \xrightarrow{\texttt{conc}} \langle e_1 \; ; \; e_2 :: k \; ; \; w \rangle \xrightarrow{n-1} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Then from the $PW\pi$ machine transitions and the induction hypothesis:

$$\langle p \; ; \; w \rangle \rightarrow \langle p_1 \; ; \; w \rangle \xrightarrow{*} \langle \texttt{null} \; ; \; \varepsilon \rangle$$

- $e = {e_1}^*$:

$$\langle {e_1}^* \; ; \; k \; ; \; w \rangle \xrightarrow{n} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle$$

Where $\pi, p \models {e_1}^*$ and $\texttt{stack}\, p = k$. Suppose $\pi, p_1 \models e_1$ so that: $\pi(p) = {p_1}^*$. From the definition of $\texttt{cont}()$, we get:

$$\texttt{cont}\, p_1 = p$$

Which in turn implies: $\texttt{stack}\, p_1 = {e_1}^* :: k$. Now from the EKW machine

transitions we have two possibilities:

$$\langle {e_1}^* \; ; \; k \; ; \; w \rangle \rightarrow \langle \varepsilon \; ; \; k \; ; \; w \rangle \xrightarrow{n-1} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle \tag{A}$$

$$\langle {e_1}^* \; ; \; k \; ; \; w \rangle \rightarrow \langle e_1 \; ; \; {e_1}^* :: k \; ; \; w \rangle \xrightarrow{n-1} \langle \varepsilon \; ; \; [\,] \; ; \; \varepsilon \rangle \tag{B}$$

The proof for the run (A) is similar to the $(e = a)$ case, and the proof for the run (B) follows a similar argument to that of case $(e = e_1 e_2)$. Finally, the proof of the reverse implication can be obtained by following a similar analysis of the $PW\pi$ run. □

3.3.1 Lockstep construction in general

As we have seen, the PWπ machine is built from two kinds of steps. Pointers can be evolved via $p \longrightarrow q$ by moving in the syntax tree without reading any input. When a node for a constant is reached, it can be matched to the first character in the input via a step $p \xrightarrow{a} q$.

Definition 3.3.10 Let $S \subseteq \mathrm{dom}(\pi) \cup \{\texttt{null}\}$ be a set of pointers. We define the evolution $\Box\, S$ of S as the following set:

$$\Box\, S = \{q \in \mathrm{dom}(\pi) \mid \exists p \in S. p \longrightarrow^* q \wedge \exists a. \pi(q) = a\}$$

Forming $\Box\, S$ is similar to computing the ε-closure in automata theory. However, this operation is not a closure operator, because $S \subseteq \Box\, S$ does not hold in general. When one computes $\Box\, S$ incrementally, elements are removed as well as added. Avoiding infinite loops by adding and removing the same element is the main difficulty in the computation.

We define a transition relation analogous to Definition 3.3.6, but as a deterministic relation on *sets* of pointers. We refer to these as macro steps,

as they assume the computation of $\Box\, S$ as given in a single step, whereas an implementation needs to compute it incrementally.

Definition 3.3.11 (Lockstep transitions) Let $S, S' \subseteq \mathrm{dom}(\pi) \cup \{\mathtt{null}\}$ be sets of pointers.

$$S \implies S' \quad \text{if } S' = \Box\, S$$

$$S \overset{a}{\implies} S' \quad \text{if } S' = \{q \in \mathrm{dom}(\pi) \mid \exists p \in S. p \xrightarrow{a} q\}$$

A set of pointers is first evolved from S to $\Box\, S$. Then, moving from a set of pointers $\Box\, S$ to S' via $\Box\, S \overset{a}{\implies} S'$ advances the state of the machine by advancing all pointers that can match a to their continuations. All other pointers are deleted as unsuccessful matches.

Definition 3.3.12 (Generic lockstep machine) The generic lockstep machine has configurations of the form $\langle S \;;\; w \rangle$. Transitions are defined using Definition 3.3.11:

$$\frac{S \overset{a}{\implies} S'}{\langle S \;;\; a\; w \rangle \Rightarrow \langle S' \;;\; w \rangle} \qquad \frac{S \implies S'}{\langle S \;;\; w \;\rangle \Rightarrow \langle S' \;;\; w \rangle}$$

Accepting states of the machine are of the form $\langle S \;;\; \varepsilon \rangle$, where $\mathtt{null} \in S$.

Lemma 3.3.13 For a heap $\pi, p \models e$ there is a run of the PWπ machine:

$$\langle p \;;\; w \rangle \xrightarrow{*} \langle \mathtt{null} \;;\; \varepsilon \rangle$$

if and only if there is a run of the lockstep machine

$$\langle \{p\} \;;\; w \rangle \Rightarrow \ldots \Rightarrow \langle S \;;\; \varepsilon \rangle$$

for some set of pointers S with $\mathtt{null} \in S$.

Proof We define a simulation relation $\sim$ between the two machines as follows:

$$\langle p \; ; \; w \rangle \sim \langle S \; ; \; w \rangle \text{ if } p \in S$$

To establish the forward implication, we analyse individual macro steps of the PWπ machine as shown below:

$$\langle p_i \; ; \; w_i \rangle \xrightarrow{*} \langle q_i \; ; \; w_i \rangle \xrightarrow{a} \langle p_{i+1} \; ; \; w_{i+1} \rangle$$

Where,

$$p_0 = p \quad w_0 = w \quad w_i = a w_{i+1}$$

$$\pi(q_i) = a \quad \texttt{cont}\, q_i = p_{i+1} \quad p_n = \texttt{null} \quad w_n = \varepsilon$$

Now, if $p_i \in S$ then it follows from Definition 3.3.11 that there is a run of the lockstep machine such that:

$$S \Longrightarrow S' \overset{a}{\Longrightarrow} S''$$

where $q_i \in S'$ and $p_{i+1} \in S''$. In other words, if $\langle p_i \; ; \; w_i \rangle \sim \langle S \; ; \; w_i \rangle$ then $\langle q_i \; ; \; w_i \rangle \sim \langle S' \; ; \; w_i \rangle$ and $\langle p_{i+1} \; ; \; w_{i+1} \rangle \sim \langle S'' \; ; \; w_{i+1} \rangle$. That is, for each macro step of the PWπ machine there is a corresponding macro step in the lockstep machine which preserves the simulation relation. Therefore, for the given run of the PWπ machine if we choose S_0 such that $\langle p_0 \; ; \; w_0 \rangle \sim \langle S_0 \; ; \; w_0 \rangle$ then there exists a corresponding run of the lockstep machine such that $\langle \texttt{null} \; ; \; \varepsilon \rangle \sim \langle S_n \; ; \; \varepsilon \rangle$.

The proof of the reverse implication can be obtained following a similar argument. □

Theorem 3.3.14 (Lockstep correctness) $e \downarrow w$ if and only if there is a

run of the lockstep machine:

$$\langle \{p\} \; ; \; w \rangle \Rightarrow \ldots \Rightarrow \langle S \; ; \; \varepsilon \rangle$$

Proof Follows from Lemmas 3.3.13, 3.3.9 and 3.2.7. □

CHAPTER 4

A STATIC ANALYSIS FOR REDOS

The previous chapter introduced an operational semantics view of regular expressions, we demonstrated how abstract machine models can be used to establish correctness properties of different matching algorithms. From this chapter onward we focus our attention on one particular problem with backtracking regular expression matchers - exponential runtime vulnerabilities.

The backtracking approach to pattern matching has been widely adopted in practice mainly to cater to the demand for more expressive pattern matching constructs (irregular expressions). As an example, leading programming language frameworks like Java, .NET, Python and Perl all provide "regex" engines which by default employ backtracking algorithms. Their adaptation also seems to be partly fueled by a lack of understanding of core computer science concepts. Such a view is expressed in a series of articles by Russ Cox [Cox07, Cox09] for example. In any case, it is quite clear that backtracking pattern matching has taken root, despite it being far inferior in performance compared to the lock-step algorithm (and its variants) discussed earlier.

The previous chapter addressed the non-termination problem of naive backtracking pattern matchers. The present chapter develops a static analysis for detecting exponential blowups. Note that unlike non-termination, exponential blowups are an inherent property of the backtracking approach, we cannot simply repair the algorithm to avoid them as we did with the infinite loops. A solution calls for more sophisticated machinery from programming language research.

4.1 Overview

For an example of an exponential blowup, consider the following regular expression:

$$(a \mid b \mid ab)^*c$$

Matching this expression against input strings of the form $(ab)^n$ leads the Java virtual machine to a halt for very moderate values of n (~ 50) on a contemporary computer. Other backtracking matchers like the PCRE library and the matcher available in the .NET platform seem to handle this particular example well. However, the ad-hoc nature of the workarounds implemented in these frameworks are easily exposed with a slightly complicated expression / input combination:

$$(a \mid b \mid ab)^*bc$$

This expression, when matched against input strings of the form $(ab)^nac$, leads to exponential blowups on all the three matchers mentioned.

The REDoS analysis builds on the idea of non-deterministic Kleene expressions. When matching the input string ab against the Kleene expression

$(a \mid b \mid ab)^*$, a match could be found by taking either of the two different paths through the corresponding NFA. If we repeat this string to form $abab$, now there are four different paths through the NFA; this process quickly builds up to an exponential amount of paths through the NFA as the pumpable string ab is repeated. A matcher based on DFAs would not face a difficulty in dealing with such expressions since the DFA construction eliminates such redundant paths. However, these expressions can be fatal for backtracking matchers based on NFAs, as their operation depends on performing a depth-first traversal of the entire search space.

We think of the various phases of the analysis as very simple and non-standard logics for judgements for different implications of the form:

$$w : p_1 \to p_2$$

Here $p_1 \to p_2$ is a proposition and w is a proof of it, which we will call its realizer. In this way, we can focus on what the analysis tries to construct, not how. Hence the analysis can be seen as a form of proof search, and it is implemented via straightforward closure algorithms.

A second use of logic or type theory in this work comes in when proving the soundness of the analysis (Chapter 5), when we need to show that the constructed string really leads to exponential runtime. While the backtracking machine that we use as an idealization of backtracking matchers (like those in the Java platform) is not very complicated, it is not straightforward to reason about how it behaves on some constructed malicious input. This is because the machine traverses the search tree in a depth-first strategy, whereas the attack string is best understood in terms of a composition of

horizontal slices of the search tree. To reason compositionally, we first introduce a calculus of search trees, inspired by substructural logics. In a nutshell, the existence of a pumpable string as part of a REDoS vulnerability amounts to the existence of a non-linear derivation in the search tree logic, essentially as in a derivation of this form:

$$\frac{p}{p, p}$$

Thus we can reason about the exponential growth of the search tree in a compositional, logical style, separate of the search strategy of the backtracking matcher. The exponential runtime of the machine then follows due to the fact that the runtime is at least the width of the search tree if the matcher is forced to explore the whole tree.

Chapter outline

Section 4.2 presents some required background on regular expression matching in a form that will be convenient for our purpose. We then define the three phases (prefix, pumping, and suffix construction) of our REDoS analysis in Section 4.3 and validate it on some examples in Section 4.4.

4.2 Basic constructs

This section presents some background material that will be needed for the analysis, such as non-deterministic automata. Figure 4.1 gives an overview of notation. We assume that the regular expression has been converted into an automaton following one of the standard constructions.

a, b, c	input symbols
w, x, y, z	strings of input symbols
p, q	NFA nodes or states
β, θ	ordered sequences $p_1 \ldots p_n$ of NFA states
Φ	sets $\{p_1, \ldots, p_n\}$ of NFA states
σ	sequences of state/index pairs (p, j)
ε	empty word or sequence
$\twoheadrightarrow$	NFA transition relation (Definition 4.2.2)
$\mapsto$	ordered NFA transition function (Definition 4.2.1)
$\rightsquigarrow$	transitions of backtracking machine (Definition 4.2.3)
$\Rrightarrow$	multistate transition function (Definition 4.2.6)
$\overset{\ell}{\mapsto}$	ordered multistate transition function (Definition 4.3.2)

Figure 4.1: Notational conventions

4.2.1 Backtracking and the ordered NFA

The usual text-book definitions of NFAs do not impose any ordering on the transition function. For an example, a traditional NFA for the regular expression $a(bc \mid bd)$ would not prioritize any of the two transitions available for character b over the other. Since backtracking matchers follow a greedy left-to-right evaluation of alternations, the alternation operator effectively becomes non-commutative in their semantics for regular expressions. Capturing this aspect in the analysis requires a specialized definition of NFAs.

If we are only concerned about acceptance, Kleene star is idempotent and alternation is commutative. If we are interested in exponential runtime, they are not. The non-commutativity of alternation is not that surprising in terms of programming language semantics, as Boolean operators like `&&` in C or `andalso` in ML have a similar semantics: first the left alternative is evaluated, and if that does not evaluate to true, the right alternative is

evaluated. Since in our tool the NFA is constructed from the syntax tree, the order is already available in the data structures. The children of a NFA node have a left-to-right ordering.

Definition 4.2.1 (Ordered NFA) An ordered NFA $\mathcal{N}$ consists of a set of states, an initial state p_0, a set of accepting states **Acc** and for each input symbol a a transition function from states to sequences of states. We write this function as

$$a : p \mapsto q_1 \ldots q_n$$

For each input symbol a and current NFA state p, we have a sequence of successor states q_i. The order is significant, as it determines the order of backtracking.

In the textbook definition of an ε-free NFA, the NFA has a transition function δ of type

$$\delta : (Q \times \Sigma) \to 2^Q$$

where Q is the set of states and Σ the set of input symbols. Here we have imposed an order on the sets in the image of the function, replacing 2^Q by Q^*, curried the function, and swapped the order of Q and Σ.

$$\begin{array}{ccccc} \Sigma & \to & (Q & \to & Q^*) \\ a & \mapsto & p & \mapsto & q_1 \ldots q_n \end{array}$$

Definition 4.2.2 The nondeterministic transition relation of the NFA is given by the following inference:

$$\frac{a : p \mapsto q_1 \ldots q_n}{a : p \twoheadrightarrow q_i}$$

Note however, that we cannot recover the ordering of the successor states

q_i from the non-deterministic transition relation. In this regard, the NFA on which the matcher is based has a little extra structure compared to the standard definition of NFA in automata theory. If we know that

$$a : p \twoheadrightarrow q_1 \text{ and } a : p \twoheadrightarrow q_2$$

we cannot decide whether the ordered transition is

$$a : p \mapsto q_1\, q_2 \text{ or } a : p \mapsto q_2\, q_1$$

To complement the ordered NFA, we introduce two kinds of data structures: ordered multistates β are finite sequences of NFA states p, where the order is significant. Multistates Φ represent sets of NFA states, so they can be represented as lists, but are identified up to reordering. Each ordered multistate β can be turned into a multistate given by the set of its elements. We write this set as $\mathsf{Set}(\beta)$. If

$$\beta = p_1 \ldots p_n$$

then

$$\mathsf{Set}(\beta) = \{p_1, \ldots, p_n\}$$

The difference between β and $\mathsf{Set}(\beta)$ may appear small, but the notion of equality for sets is less fine-grained than for sequences, which has an impact on the search space that the analysis has to explore.

4.2.2 The abstract machines

The analysis assumes exact matching semantics of regular expressions. Given regular expression e and the input string w, the matcher is required to find a match of the entire string, as opposed to a sub-string. Most practical

matchers search for a sub-match by default. However, such behavior can be modeled in exact matching semantics by augmenting the regular expression with match-all constructs at either end of the expression, as in $(.^*e.^*)$. Practical implementations offer special "anchoring" constructs that allow regular expression authors to enforce exact matching semantics. For an example, expressions of the form $(\hat{}e\$)$ require them to be matched against the entire input string.

While the theoretical formulation of our analysis assumes exact matching semantics (thus avoiding unnecessary clutter), our implementation assumes sub-match semantics, since it is more useful in practice. The translation between the two semantics is quite straightforward.

Definition 4.2.3 (Backtracking abstract machine) Given an ordered NFA, the backtracking machine is defined as follows. We assume an input string w as given. Machine transitions may depend on w, but it does not change during transitions, so that we do not explicitly list it as part of the machine state. The input symbol at position j in w is written as $w[j]$ (j is 0-based).

- States of the backtracking machine are finite sequences of the form

  $$\sigma = (p_0, j_0) \ldots (p_n, j_n)$$

  where each of the p_i is an NFA state and each of the j_i is an index into the current input string. We refer to individual (p, i) pairs as frames (as in stack frames).
- The initial state of the machine is the sequence of length 1 containing the frame:

  $$(p_0, 0)$$

- The machine has matching transitions, which are inferred from the transition function of the ordered NFA as follows:

$$\frac{w[j] = a \qquad a : p \mapsto q_1 \ldots q_n}{w \Vdash (p, j)\, \sigma \rightsquigarrow (q_1, j+1) \ldots (q_n, j+1)\, \sigma}$$

- The machine has failing transitions, of the form

$$(p, j)\ \sigma \rightsquigarrow \sigma$$

  where $w[j] \neq a$ or j is the length of w and $p \notin \mathbf{Acc}$.

- Accepting states are of the form:

$$w \Vdash (p, j)\ \sigma$$

  where $p \in \mathbf{Acc}$ and j is the length of w.

- Transition sequences in n steps are written as $\overset{n}{\rightsquigarrow}$ and inferred using the following rules:

$$\frac{w \Vdash \sigma \rightsquigarrow \sigma'}{w \Vdash \sigma \overset{1}{\rightsquigarrow} \sigma'} \qquad \frac{w \Vdash \sigma_1 \overset{n}{\rightsquigarrow} \sigma_2 \qquad w \Vdash \sigma_2 \overset{m}{\rightsquigarrow} \sigma_3}{w \Vdash \sigma_1 \overset{n+m}{\rightsquigarrow} \sigma_3}$$

  We write $w \Vdash \sigma_1 \overset{*}{\rightsquigarrow} \sigma_2$ for $\exists n. w \Vdash (\sigma_1 \overset{n}{\rightsquigarrow} \sigma_2)$.

- Final states are either accepting or the empty sequence.

The state of the backtracking machine is a stack that implements failure continuations. When the state is of the form $(p, j)\sigma$, the machine is currently trying to match the symbol at position j in state p. Should this match fail, it will pop the stack and proceed with the failure continuation σ.

Example 4.2.4 Let $e = ((a\,a) \mid (a\,c))$. The NFA is given by

$$\begin{aligned} a &: p_0 \mapsto p_1\ p_2 \\ a &: p_1 \mapsto p_3 \\ c &: p_2 \mapsto p_4 \\ \mathbf{Acc} &= \{p_3, p_4\} \end{aligned}$$

Let the input be $w = a\,c$. The machine matches it as follows:

$$\begin{aligned} a\,c &\Vdash (p_0, 0) \\ &\rightsquigarrow (p_1, 1)\ (p_2, 1) \\ &\rightsquigarrow (p_2, 1) \\ &\rightsquigarrow (p_4, 2) \end{aligned}$$

The backtracking machine definition leaves a lot of leeway to the implementation. Implementation details are abstracted in the ordered transition relation. The most important choice in the definition is that the machine performs a depth-first traversal of the search tree. In principle, a backtracking matcher could also use breadth-first search. In that case, our REDoS analysis would not be applicable, and such matchers may avoid exponential run-time. However, the space requirements of breadth-first search are arguably prohibitive. A more credible alternative to backtracking matchers is Thompson's matcher [Tho68, Cox07, Cox09], which is immune to REDoS. As noted previously however, most practical implementations have opted for the backtracking approach due to the inflexibility of the lockstep algorithm (when supporting extended, non-regular pattern matching constructs). The REDoS problem in the backtracking matcher therefore remains quite signif-

icant.

Definition 4.2.5 (Lockstep abstract machine) The lockstep abstract machine, based on Thompson's matcher, is defined as follows:

- The states of the lockstep matcher are of the form

  $$(\Phi, j)$$

  where Φ is a set of NFA states and j is an index into the input string w.

- The initial state is

  $$(\{p_0\}, 0)$$

- The matching transition are inferred as follows:

  $$\frac{w[j] = a \qquad a : p_1 \mapsto \beta_1 \quad \ldots \quad a : p_n \mapsto \beta_n}{w \Vdash (\{p_1, \ldots, p_n\}, j) \rightsquigarrow (\mathsf{Set}(\beta_1) \cup \ldots \cup \mathsf{Set}(\beta_n), j + 1)}$$

- An accepting state is of the form

  $$(\Phi, j)$$

  where j is the length of w and $\Phi \cap \mathbf{Acc} \neq \emptyset$.

At each step, redundancy elimination is performed by taking sets rather than sequences.

The lockstep machine will not be used in the rest of the discussion. It is only presented here to illustrate how it avoids the state-space explosion through redundancy elimination.

4.2.3 The power DFA construction

Based on a construction that is standard in automata theory and compiler construction, for each NFA there is a DFA. The set of states of this DFA is the powerset of the set of states of the NFA. We refer to such sets of NFA states as multistates.

Definition 4.2.6 (Power DFA) Given an NFA, its power DFA is constructed as follows:

- The states of the power DFA are sets Φ of NFA states.
- The transition relation $\Rrightarrow$ is defined as

  $$a : \Phi_1 \Rrightarrow \Phi_2$$

  if and only if

  $$\Phi_2 = \{p_2 \mid \exists p_1 \in \Phi_1 . a : p_1 \twoheadrightarrow p_2\}$$

- The initial state of the power DFA is the singleton set $\{p_0\}$.
- The accepting states of the power DFA are those sets Φ for which $\Phi \cap \mathbf{Acc} \neq \emptyset$.

Definition 4.2.7 The transition function of the power DFA is extended from strings w to sets of strings W using the following rule

$$\frac{\Phi_2 = \{p_2 \mid \exists p_1 \in \Phi_1 . \exists w \in W . w : p_1 \twoheadrightarrow p_2\}}{W : \Phi_1 \Rrightarrow \Phi_2}$$

Intuitively, we regard W as the set of realizers that take us from Φ_1 to Φ_2. Note that it is not only the case that (elements of) W will take us from (elements of) Φ_1 to (elements of) Φ_2. Moreover, everything in Φ_2 arises this way from Φ_1 and W. In that sense, a judgement $W : \Phi_1 \Rrightarrow \Phi_2$ is stronger

than realizability or pre- and post-conditions. The fact that Φ_2 is uniquely determined by Φ_1 and W is useful for the analysis.

4.3 The REDoS analysis

For a given regular expression of the form $e_1 e_2{}^* e_3$, the analysis attempts to derive an attack string of the form:

$$xy^n z$$

The presence of a pumpable string y signals the analyser that a corresponding prefix x and a suffix z need to be derived in order to form the final attack string configuration. The requirements on the different segments of the attack string are as follows:

$$\begin{aligned} x &: x \in L(e_1) \\ y &: y \in L(e_2{}^*) \quad (\text{with } b > 1 \text{ paths}) \\ z &: xy^n z \notin L(e_1 e_2{}^* e_3) \end{aligned}$$

Intuitively, the prefix x leads a backtracking matcher to a point where it has to match the (vulnerable) Kleene expression $e_2{}^*$. At this point the matcher is presented with n ($n > 0$) copies of the pumpable string y, increasing the search space of the matcher to the order of b^n. At the end of each of the search attempts (paths through the NFA), the suffix z causes the matcher to backtrack, forcing an exploration of the entire search space.

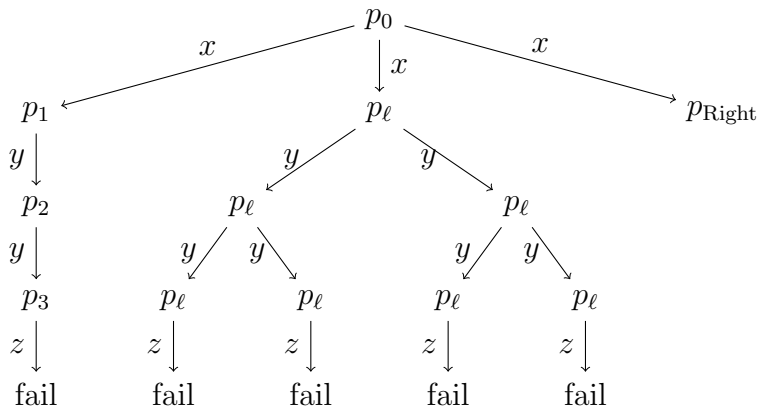


Figure 4.2: Branching search tree with left context for $x\,y\,y\,z$

4.3.1 The phases of the analysis

Overall, the REDoS analysis of a node p_ℓ (loop node) consists of three phases. The phases all work by incrementally exploring a transition relation. These relations are the power DFA transition relation $\Rrightarrow$ and a new ordered variant $\vec{\Rrightarrow}_\ell$ (Definition 4.3.2). The three analysis phases construct a REDoS prefix x, a pumpable string y and a REDoS suffix z:

$$
\begin{array}{rl}
\text{Prefix analysis} & \left\{ \begin{array}{lcll} x & : & p_0 \vec{\Rrightarrow}_\ell (\beta\, p_\ell\, \beta') & \end{array} \right. \\
\text{Pumpable analysis} & \left\{ \begin{array}{lcll} y_1 & : & \Phi_x \Rrightarrow \Phi_{y1} & \text{where } \Phi_x = \mathsf{Set}(\beta\, p_\ell) \\ a & : & \Phi_{y1} \Rrightarrow \Phi_{y1a} & \\ y_2 & : & \Phi_{y1a} \Rrightarrow \Phi_{y2} & \text{where } \Phi_{y2} \subseteq \Phi_x \end{array} \right. \\
\text{Suffix analysis} & \left\{ \begin{array}{lcll} z & : & \Phi_{y2} \Rrightarrow \Phi_{\text{fail}} & \text{where } \Phi_{\text{fail}} \cap \mathbf{Acc} = \emptyset \end{array} \right.
\end{array}
$$

4.3.2 Prefix analysis

The analysis needs to find a string that causes the matcher to reach p_ℓ. However, due to the nondeterminism of the underlying NFA, it is not enough to check reachability. The same string x could also lead to some other states before p_ℓ is reached by the matcher. If one of these states could lead to acceptance, the matcher will terminate successfully, and p_ℓ will never be reached. In this case, there is no vulnerability, regardless of any exponential blowup in the subtree under p_ℓ. See Figure 4.2.

Definition 4.3.1 The operator $\ggg$ removes all but the leftmost occurrences in sequences according to the following rules:

$$
\begin{array}{lcll}
[\,] & \ggg & [\,] & \\
p\,\beta & \ggg & p\,\beta' & \text{if } (\nexists \beta_1, \beta_2 \,.\, \beta = \beta_1\, p\, \beta_2) \wedge \beta \ggg \beta' \\
p\,\beta & \ggg & \beta' & \text{if } (\exists \beta_1, \beta_2 \,.\, \beta = \beta_1\, p\, \beta_2) \wedge p\,\beta_1\,\beta_2 \ggg \beta'
\end{array}
$$

Note that $\ggg$ is applied on shorter sequences on the R.H.S, ensuring termination. Moreover, in each reduced sequence each p can appear at most once, so there are only finitely many sequences that can be reached in the REDoS prefix analysis (below).

Definition 4.3.2 Let p_ℓ be the NFA state we are currently analyzing. The transition relation $\overset{\curvearrowright}{\vdash}_\ell$ for ordered multistates is defined as follows:

$$
\frac{\beta_1 = (p_1 \ldots p_n) \qquad a : p_i \mapsto \theta_i \qquad (\theta_1 \ldots \theta_n) \ggg \beta_2}{a : \beta_1 \overset{\curvearrowright}{\vdash}_\ell \beta_2}
$$

The relation is extended to strings:

$$\frac{w : \beta_1 \Rsh_\ell \beta_2 \qquad a : \beta_2 \Rsh_\ell \beta_3}{(w\,a) : \beta_1 \Rsh_\ell \beta_3} \qquad \frac{}{\varepsilon : \beta \Rsh_\ell \beta}$$

The REDoS prefix analysis computes all ordered multistates β reachable from p_0, together with a realizer w, using the following rules:

$$\frac{}{(\varepsilon, p_0) \in \mathcal{R}} \qquad \frac{(w, \beta_1) \in \mathcal{R} \quad a : \beta_1 \Rsh_\ell \beta_2 \quad \nexists w'.(w', \beta_2) \in \mathcal{R}}{(w\,a, \beta_2) \in \mathcal{R}}$$

In the implementation, we keep a set $\mathcal{R}$. It is initialized to (ε, p_0). We then repeatedly check if there is a (w, β_1) in the set such that for some a there is a transition $a : \beta_1 \Rsh_\ell \beta_2$. If there is, we add $(w\,a, \beta_2)$ to $\mathcal{R}$ and repeat the process. We terminate when no new β_2 has been found in the last iteration. Finally, the analysis isolates (w, β) pairs of the form $(x, \beta\ p_\ell\ \beta')$ and takes Φ_x as $\mathsf{Set}(\beta\ p_\ell)$ for each such pair.

4.3.3 Pumping analysis

Definition 4.3.3 A branch point is a tuple

$$(p_N, a, \{p_{N1}, p_{N2}\})$$

such that $p_{N1} \neq p_{N2}$, $a : p_N \twoheadrightarrow p_{N1}$ and $a : p_N \twoheadrightarrow p_{N2}$.

For example, if p_N has three successor nodes p_1, p_2 and p_3 for the same input symbol a, there are three different branch points:

$$(p_N, a, \{p_1, p_2\})$$

$$(p_N, a, \{p_1, p_3\})$$

$$(p_N, a, \{p_2, p_3\})$$

There can be only finitely many non-deterministic nodes in the given NFA. For each of them, we need to solve a reachability problem.

The pumping analysis can be visualized with the diagram in Figure 4.3. The analysis aims to find two different paths leading from p_ℓ to itself. Such paths must at some point include a nondeterministic node p_N that has at least two transitions to different nodes p_{N1} and p_{N2} for the same symbol a. For such a node to lie on a path from p_ℓ to itself, there must be some path labeled y_1 leading from p_ℓ to p_N, and moreover there must be paths from the two child nodes p_{N1} and p_{N2} leading back to p_ℓ, such that both these paths have the label y_2. The left side of Figure 4.3 depicts this situation.

So far we have only considered what states *may* be reached. Due to the nondeterminism of the transition relation $a : p \twoheadrightarrow q$, there may be other states that can be reached for the same strings y_1 and y_2. For an example, the regular expression $(.^* \mid e)$ is not vulnerable even if e itself is vulnerable; the expression on the left will always yield a successful match for any string we generate (the meta-character "." matches any input symbol). Therefore, we also need to perform a *must* analysis that keeps track of all the states reachable via the strings we construct. This analysis uses the transition relation $\Rrightarrow$ of the power DFA between sets of NFA states. In Figure 4.3, it is shown on the right-hand side.

Intuitively, we run the two transition relations in parallel on the same input string. More formally, this involves constructing a product of two relations. Before we reach the branching point, we run the relations $\twoheadrightarrow$ and $\Rrightarrow$ in parallel. After the nondeterministic node p_N has produced two different successors, we need to run two copies of $\twoheadrightarrow$ in parallel with $\Rrightarrow$. One may

visualize this situation by reading the diagram in Figure 4.3 horizontally: above the splitting at p_N, there are two arrows in parallel for y_1, whereas below that node, there are three arrows in parallel for a and y_2.

The twofold transition relation $\twoheadrightarrow_{\mathbf{2}}$ for running $\twoheadrightarrow$ in parallel with $\Rrightarrow$ is given by the rules in Figure 4.4. Analogously, the threefold product transition relation $\twoheadrightarrow_{\mathbf{3}}$ for running two copies of $\twoheadrightarrow$ in parallel with $\Rrightarrow$ is given by the rules in Figure 4.5.

In summary, the pumping analysis consists of two phases:

1. Given p_ℓ and Φ_x, the analysis searches for a realizer y_1 for reaching some nondeterministic node p_N:

$$y_1 : (p_\ell, \Phi_x) \twoheadrightarrow_{\mathbf{2}} (p_N, \Phi_{y1})$$

2. Given the successor nodes p_{N1} and p_{N2} of some p_N node, the analysis searches for a realizer y_2 for reaching p_ℓ:

$$y_2 : (p_{N1}, p_{N2}, \Phi_{y1a}) \twoheadrightarrow_{\mathbf{3}} (p_\ell, p_\ell, \Phi_{y2})$$

   Moreover, the analysis checks that the constructed state Φ_{y2} satisfies the inclusion:

$$\Phi_{y2} \subseteq \Phi_x$$

If both phases of the analysis succeed, the string $y_1\, a\, y_2$ is returned as the pumpable string, together with the state Φ_{y2}.

Example 4.3.4 The following diagram shows an NFA corresponding to the regular expression $(a|b|ab)^*$:

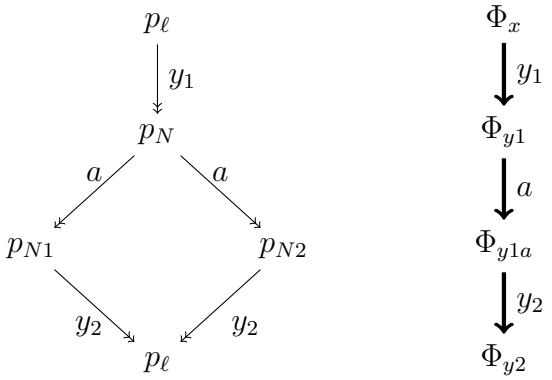
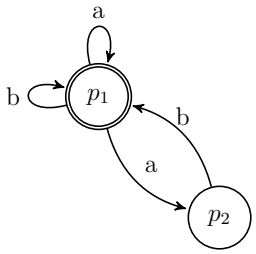


Figure 4.3: Pumping analysis construction of $y_1\, a\, y_2$: "may" on the left using $\twoheadrightarrow$, and "must" on the right using $\Rrightarrow$



Taking $p_\ell = p_1$ and $\Phi_x = \{p_1\}$, the pumping analysis leads to the following derivation:

$$\begin{aligned} y_1 = \varepsilon \qquad & \varepsilon : (p_1, \{p_1\}) \twoheadrightarrow_{\mathbf{2}} (p_1, \{p_1\}) \\ (p_1, a, \{p_1, p_2\}) \qquad & a : \{p_1\} \Rrightarrow \{p_1, p_2\} \\ y_2 = b \qquad & b : (p_1, p_2, \{p_1, p_2\}) \twoheadrightarrow_{\mathbf{3}} (p_1, p_1, \{p_1, p_2\}) \end{aligned}$$

Here we have an *unstable* derivation since $\{p_1, p_2\} \not\subseteq \{p_1\}$ (i.e. $\Phi_{y_2} \not\subseteq \Phi_x$). If we were to take $\Phi_x = \{p_1, p_2\}$ (i.e. $x = a$), the resulting derivation would be stable (for the same pumpable string ab).

$$\frac{w : (p_1, \Phi_1) \twoheadrightarrow_{\mathbf{2}} (p_2, \Phi_2) \qquad \begin{array}{ll} b : & p_2 \twoheadrightarrow p_3 \\ b : & \Phi_2 \Rrightarrow \Phi_3 \end{array}}{(w\, b) : (p_1, \Phi_1) \twoheadrightarrow_{\mathbf{2}} (p_3, \Phi_3)}$$

$$\frac{}{\varepsilon : (p, \Phi) \twoheadrightarrow_{\mathbf{2}} (p, \Phi)}$$

Figure 4.4: The twofold product transition relation $\twoheadrightarrow_{\mathbf{2}}$

$$\frac{w : (p_1, p_1', \Phi_1) \twoheadrightarrow_{\mathbf{3}} (p_2, p_2', \Phi_2) \qquad \begin{array}{ll} b : & p_2 \twoheadrightarrow p_3 \\ b : & p_2' \twoheadrightarrow p_3' \\ b : & \Phi_2 \Rrightarrow \Phi_3 \end{array}}{(w\, b) : (p_1, p_1', \Phi_1) \twoheadrightarrow_{\mathbf{3}} (p_3, p_3', \Phi_3)}$$

$$\frac{}{\varepsilon : (p, p', \Phi) \twoheadrightarrow_{\mathbf{3}} (p, p', \Phi)}$$

Figure 4.5: The threefold product transition relation $\twoheadrightarrow_{\mathbf{3}}$

4.3.4 Suffix analysis

For each Φ_{y2} constructed by the pumping analysis, the REDoS failure analysis computes all multistates Φ_{fail} such that there is a z with

$$z : \Phi_{y2} \Rrightarrow \Phi_{\text{fail}} \qquad \wedge \qquad \Phi_{\text{fail}} \cap \mathbf{Acc} = \emptyset$$

Intuitively, z fails *all* the states in Φ_{y2} by taking them to Φ_{fail}, which does not contain any accepting states.

4.4 Test cases for the REDoS analysis

In order to demonstrate the behavior of our improved analyser, here we present examples that exercise the most important aspects of its operation.

4.4.1 Non commutativity of alternation

This aspect of the analysis can be illustrated with the following two example expressions:

$$.^* \mid (a \mid b \mid ab)^*c$$

$$(a \mid b \mid ab)^*c \mid .^*$$

Even though the two expressions correspond to the same language, only the second expression yields a successful attack. In the first expression, all the multi-states starting from Φ_{x_1} ($\mathsf{Set}(\beta p_\ell)$) consist of a state corresponding to the expression (.*), which implies that this expression is capable of consuming any input string without invoking the vulnerable Kleene expression. On the other hand, Φ_{x_1} calculated for the second expression lacks a state corresponding to (.*), leading to the following attack string configuration:

$$x = \varepsilon \quad y = ab \quad z = \varepsilon$$

4.4.2 Prefix construction

Prefix construction plays one of the most crucial roles in finding an attack string. In the following example, only a certain prefix leads to a successful attack string derivation:

$$c.^*|(c \mid d)(a \mid b \mid ab)^*e$$

Notice that a prefix c would trigger the (.*) on the left due to the left-biased treatment of alternation in backtracking matchers. The prefix d on the other hand forces the matcher out of this possibility. The difference between these

two prefixes is captured in two different values of (x, Φ_x):

$$(c, \{p_1, p_2\}) \quad (d, \{p_2\})$$

Where

$$p_1 \models .^* \text{ and } p_2 \models (a \mid b \mid ab)^* e$$

Only the latter of these two leads to a successful attack string:

$$x = d \quad y = ab \quad z = \varepsilon$$

Prefix construction may also lead to loop unrolling when necessary. For an example, consider the following regex:

$$(a \mid b).^* | c^* (a \mid ab \mid b)^* d$$

Without the unrolling of the Kleene expression c^*, any pumpable string intended for the vulnerable Kleene expression will be consumed by the alternation on the left. The analyser captures this situation again as two different values of (x, Φ_x), one for $x = c$ and the other for either $x = a$ or $x = b$. Only the former value leads to a successful attack string:

$$x = c \quad y = ab \quad z = \varepsilon$$

The amount of loop unrolling is limited by the finite-ness of the Φ_x values. In the following example, the loop c^* needs to be unrolled twice:

$$(c \mid a \mid b)(a \mid b).^* | c^* (a \mid b \mid ab)^* d$$

Here, unrolling c^* 0 - 2 times leads to three distinct values of Φ_x due to the different matching states on the left alternation. Only one of those unrollings leads to a successful attack string:

$$x = cc \quad y = ab \quad z = \varepsilon$$

4.4.3 Pumpable construction

As is the case with prefixes, the existence of an attack string may depend on the construction of an appropriate pumpable string. For an example, consider the following regex:

$$(a \mid a \mid b \mid b)^*(a.^* \mid c)$$

Here the pumpable string a does not yield an attack string since it also triggers the $(.^*)$ continuation. On the other hand, the pumpable string b avoids this situation and leads to the following attack string configuration:

$$x = \varepsilon \quad y = b \quad z = \varepsilon$$

Similar to the prefix analysis, pumpable analysis utilises (y, Φ_y) values to select between pumpable strings.

In some cases, the pumpable construction overlaps with prefix construction. In the example below, an attack string may be composed in two different ways:

$$d.^*|((c \mid d)(a \mid a))^*b$$

Here, choosing ca as the pumpable string leads to a successful attack string derivation:

$$x = \varepsilon \quad y = ca \quad z = \varepsilon$$

However, it is also possible to form an attack string with the following configuration:

$$x = ca \quad y = da \quad z = \varepsilon$$

The important point here is that the attack string must begin with a c instead of a d in order to avoid the obvious match on the left. The analyser is capable

of finding both the configurations that meet this requirement.

Pumpable construction may also lead to loop unrolling when necessary, as demonstrated by the following example:

$$a.^*|(c^*a(b \mid b))^*d$$

Without unrolling the inner loop c^*, the pumpable string ab would trigger the alternation on the left. A successful attack string requires the unrolling of this inner loop, as in the following configuration:

$$x = \varepsilon \quad y = cab \quad z = \varepsilon$$

As with the previous example, the unrolling of the inner loop c^* may be performed as part of the prefix construction, leading to the following alternate attack string configuration:

$$x = cab \quad y = ab \quad z = \varepsilon$$

The latter configuration may be considered more desirable in that it makes the the pumpable string shorter, leading to much smaller attack strings.

CHAPTER 5

CORRECTNESS OF THE ANALYSIS

Previous chapter presented our static analysis for exponential runtime explosions in backtracking regular expression matchers, we made informal arguments about its correctness by validating it against several example regular expressions. This chapter presents the core theoretical result of our thesis; soundness and completeness of the analysis. In other words, we construct mathematical proofs which establish that the analysis always produces correct results, and that it is capable of finding any exponential runtime vulnerability present within a given regular expression.

5.1 Soundness of the analysis

The backtracking machine performs a depth-first search of a search tree. Proofs about runs of the machine are thus complicated by the fact that the construction of the tree and its traversal are conflated. To make reasoning more compositional, we define a substructural calculus for constructing search trees. Machine runs correspond to paths from roots to leaves in these

trees.

5.1.1 Search tree logic

Definition 5.1.1 (Search tree logic) The search tree logic has judgements of the form

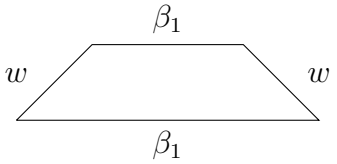
$$w : \beta_1 \triangle \beta_2$$

where w is an input string, and both β_1 and β_2 are sequences of NFA states. The inference rules are given in Figure 5.1.

Intuitively, the judgement

$$w : \beta_1 \triangle \beta_2$$

means that there is a horizontal slice of the search tree, such that the nodes at the top form the sequence β_1, the nodes at the bottom form the sequence β_2, and all paths have the same sequence of labels, forming w:



Each w represents an NFA run $w : p_1 \twoheadrightarrow p_2$ for some p_1 that occurs in β_1 and some p_2 that occurs in β_2. The string w labels the sides of the trapezoid, since that determines the compatible boundary for parallel composition. Again we may like to think of w as a proof of reachability. Here the reachability is not in the NFA, but in the matcher based on it.

The trapezoid can be stacked on top of each other if they share a common β at the boundary. They can be place side-by-side if they have the same w

$$\frac{a : p \mapsto \beta}{a : p \triangle \beta}\,(\textsc{Trans1}) \qquad \frac{\nexists \beta.a : p \mapsto \beta}{a : p \triangle \varepsilon}\,(\textsc{Trans2})$$

$$\frac{w_1 : \beta_1 \triangle \beta_2 \qquad w_2 : \beta_2 \triangle \beta_3}{(w_1\, w_2) : \beta_1 \triangle \beta_3}\,(\textsc{SeqComp})$$

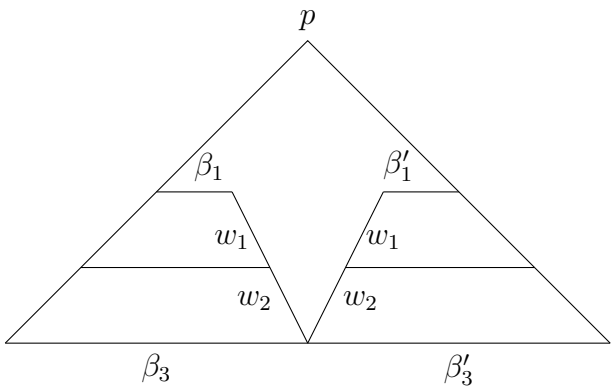
$$\frac{}{\varepsilon : \beta \triangle \beta}\,(\varepsilon\textsc{Seq})$$

$$\frac{w : \beta_1 \triangle \beta_2 \qquad w : \beta_1' \triangle \beta_2'}{w : (\beta_1\, \beta_1') \triangle (\beta_2\, \beta_2')}\,(\textsc{ParComp})$$

$$\frac{}{w : \varepsilon \triangle \varepsilon}\,(\varepsilon\textsc{Par})$$

Figure 5.1: Search tree logic

on the inside:



5.1.2 Pumpable implies exponential tree growth

We use the search tree logic to construct a tree by closely following the phases of our REDoS analysis. The exponential growth of the search tree in response to pumping is easiest to see when thinking of horizontal slices across the search tree for each pumping of y. The machine computes a diagonal cut across the search tree as it moves towards the left corner. The analysis constructs horizontal cuts with all states at the same depth. It is sufficient

to show that the width of the search tree grows exponentially. The width is easier to formalize than the size.

We need a series of technical lemmas connecting different transition relations:

Lemma 5.1.2 The following rule is admissible:

$$\frac{w : \Phi_1 \Rrightarrow \Phi_2 \qquad w : \Phi_1' \Rrightarrow \Phi_2'}{w : (\Phi_1 \cup \Phi_1') \Rrightarrow (\Phi_2 \cup \Phi_2')}$$

Lemma 5.1.3 ($\Rrightarrow \triangle$ simulation) If $w : \Phi_1 \Rrightarrow \Phi_2$, $w : \beta_1 \triangle \beta_2$ and $\Phi_1 = \mathsf{Set}(\beta_1)$, then $\Phi_2 = \mathsf{Set}(\beta_2)$.

Proof Suppose:

$$\beta_1 = (p_1 \ldots p_n) \qquad a : p_i \mapsto \theta_i$$

Then from the search tree logic we get $a : \beta_1 \triangle (\theta_1 \ldots \theta_n)$. Moreover, the definition of $\Rrightarrow$ implies $a : \{p_i\} \Rrightarrow \mathsf{Set}(\theta_i)$. Now, applying Lemma 5.1.2 gives:

$$a : \mathsf{Set}(\beta_1) \Rrightarrow \mathsf{Set}(\theta_1) \cup \ldots \cup \mathsf{Set}(\theta_n) = \mathsf{Set}(\theta_1 \ldots \theta_n)$$

Therefore, the result holds for strings of unit length. An induction on the length of w completes the proof. $\square$

Lemma 5.1.4 ($\vec{\Gamma}_\ell \triangle$ simulation) If $w : \beta_1 \vec{\Gamma}_\ell \beta_2$, $w : \beta_1' \triangle \beta_2'$ and $\beta_1' \ggg \beta_1$, then $\beta_2' \ggg \beta_2$.

Proof Suppose:

$$\beta_1' = (p_{11} \ldots p_{mk}) \qquad a : p_{ij} \mapsto \theta_{ij}$$

Where p_{ij} corresponds to the jth occurrence of the pointer p_i. Equivalently:

$$p_{ij} = p_{i'j'} \Longleftrightarrow i = i'$$

Given $\beta_1' \ggg \beta_1$, we deduce:

$$(p_{11} \ldots p_{mk}) \ggg (p_{11} \ldots p_{m1}) = \beta_1$$

Now, given $a : \beta_1 \,\vec{\Gamma}_\ell\, \beta_2$, the definition of $\vec{\Gamma}_\ell$ gives:

$$(\theta_{11} \ldots \theta_{m1}) \ggg \beta_2$$

On the other hand, $a : \beta_1' \triangle \beta_2'$ gives:

$$\beta_2' = (\theta_{11} \ldots \theta_{mk})$$

The definition of $\ggg$ can be generalized for multi-states, which leaves us with:

$$(\theta_{11} \ldots \theta_{mk}) \ggg (\theta_{11} \ldots \theta_{m1})$$

That is, we have shown:

$$\beta_2' = (\theta_{11} \ldots \theta_{mk}) \ggg (\theta_{11} \ldots \theta_{m1}) \ggg \beta_2$$

An induction on the length of w completes the proof. □

Lemma 5.1.5 ($\twoheadrightarrow \triangle$ simulation) Given $w : p \twoheadrightarrow q$, there are sequences of states β_1 and β_2 such that $w : p \triangle \beta_1\, q\, \beta_2$.

Proof The base case ($w = a$) holds from the definition of $\triangle$. For the inductive step, suppose $w : p \twoheadrightarrow q$ and $a : q \mapsto q'$. Then from the induction hypothesis we get $w : p \triangle \beta_1\, q\, \beta_2$ for some β_1, β_2. Moreover, from the base case we have $a : q \triangle \beta_3\, q'\, \beta_4$ for some β_3, β_4. Assuming $a : \beta_1 \triangle \beta_1'$ and $a : \beta_2 \triangle \beta_2'$ for some β_1', β_2', the definition of $\triangle$ gives $wa : p \triangle \beta_1'\, \beta_3\, q'\, \beta_4\, \beta_{2'}$. □

Lemma 5.1.6 (Pumpable realizes non-linearity) Let y be pumpable for

some node p_ℓ. Then there exist β_1, β_2, β_3 such that:

$$y : p_\ell \triangle \beta_1\, p_\ell\, \beta_2\, p_\ell\, \beta_3$$

Proof The pumpable analysis generates a string of the form:

$$y = y_1\, a\, y_2$$

Where

$$y_1 : p_\ell \twoheadrightarrow p_N$$

$$a : p_N \mapsto (\beta\ p_{N1}\ \beta'\ p_{N2}\ \beta'')$$

$$y_2 : p_{N1} \twoheadrightarrow p_\ell \qquad y_2 : p_{N2} \twoheadrightarrow p_\ell$$

Now, Lemma 5.1.5 leads to the desired result. □

Lemma 5.1.7 Let x, y be constructed from the prefix analysis and the pumpable analysis such that:

$$\begin{array}{rcll} x & : & p_0 \Rsh_\ell\ (\beta\ p_\ell\ \beta') & \\ y & : & \mathsf{Set}(\beta\, p_\ell) \Rrightarrow \Phi_y & \Phi_y \subseteq \mathsf{Set}(\beta\, p_\ell) \end{array}$$

Then the following holds for any natural number n:

$$\Phi_{y^n} \subseteq \Phi_{y^{n-1}}$$

Where $\Phi_{y^0} = \mathsf{Set}(\beta\, p_\ell)$ and $y^n : \Phi_{y^0} \Rrightarrow \Phi_{y^n}$.

Proof By induction on n. Note that the base case ($n = 1$) holds by construction. For the inductive step, suppose $\exists\, q \in \Phi_{y^n}$, then from the definition of Φ_{y^n} we get $\exists\, p \in \Phi_{y^{n-1}}\ .\ y : p \twoheadrightarrow q$. Moreover, the induction hypothesis gives $\Phi_{y^{n-1}} \subseteq \Phi_{y^{n-2}}$. Therefore, we have $p \in \Phi_{y^{n-2}}$, which in turn implies $q \in \Phi_{y^{n-1}}$. □

The importance of Lemma 5.1.7 is that it allows us to calculate a failure

suffix z independent of the number of pumping iterations; Φ_{y^n} can only shrink as n increases.

Lemma 5.1.8 (Exponential tree growth) Let x, y, z be constructed from the analysis such that:

$$\begin{array}{rcll} x & : & p_0 \,\vec{\Gamma}_\ell\, (\beta\, p_\ell\, \beta') & \\ y & : & \mathsf{Set}(\beta\, p_\ell) \Rrightarrow \Phi_y & \Phi_y \subseteq \mathsf{Set}(\beta\, p_\ell) \\ z & : & \Phi_y \Rrightarrow \Phi_{\text{fail}} & \Phi_{\text{fail}} \cap \mathbf{Acc} = \emptyset \end{array}$$

Then there exists β_L, β_R such that:

$$x : p_0 \mathrel{\triangle} \beta_L\, p_\ell\, \beta_R \wedge \mathsf{Set}(\beta_L) = \mathsf{Set}(\beta) \tag{A}$$

$$y^n : \beta_L\, p_\ell \mathrel{\triangle} \beta_n \Rightarrow |\beta_n| \geq 2^n \tag{B}$$

$$z : \mathsf{Set}(\beta_n) \Rrightarrow \Phi' \Rightarrow \Phi' \cap \mathbf{Acc} = \emptyset \tag{C}$$

Proof

- **Statement (A)**: Suppose $x : p_0 \mathrel{\triangle} \beta_x$. Then from Lemma 5.1.4 it follows that $\beta_x \ggg \beta\, p_\ell\, \beta'$. That is, p_ℓ must occur in β_x. If we dissect β_x into $\beta_L\, p_\ell\, \beta_R$ such that $p_\ell \notin \mathsf{Set}(\beta_L)$, then from the definition of $\ggg$ it follows that $\mathsf{Set}(\beta_L) = \mathsf{Set}(\beta)$.
- **Statement (B)**: Follows from Lemma 5.1.6. The number of copies of p_ℓ doubles at each pumping iteration.
- **Statement (C)**: Suppose $y^n : \mathsf{Set}(\beta_L\, p_\ell) \Rrightarrow \beta'_n$. Since $\mathsf{Set}(\beta_L\, p_\ell) = \mathsf{Set}(\beta\, p_\ell)$ (statement A), Lemma 5.1.3 gives: $\mathsf{Set}(\beta_n) = \mathsf{Set}(\beta'_n)$. Now from Lemma 5.1.7 it follows that $\mathsf{Set}(\beta_n) \subseteq \Phi_y$. Since z cannot lead to a successful match from any state in Φ_y (by construction), the same should be true for $\mathsf{Set}(\beta_n)$.

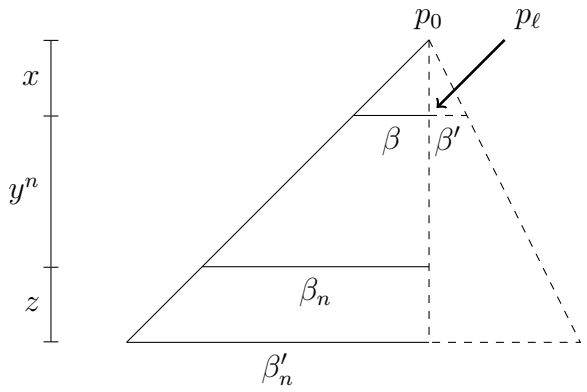


Figure 5.2: Tree growth (Lemma 5.1.8)

□

Lemma 5.1.8 may be visualized as in Figure 5.1.2. Note that the right hand slice of the tree (emanating from β') is irrelevant, the depth-first strategy of a backtracking matcher forces it to explore the left hand slice first. Since none of the states at the bottom of the tree (β'_n) are accepting, it is forced to explore the (exponentially large, $|\beta_n| \geq 2^n$) entire slice (as proved in the following section).

5.1.3 From search tree to machine runs

Having proved that the attack strings lead to exponentially large search trees, in this section we show how backtracking matchers are forced to traverse all of it. We use the notation $w[i \ldots j]$ to represent the substring of w starting at index i (inclusive) and ending at index j (exclusive).

Lemma 5.1.9 Let w be an input string of length n, s a constant offset into

w and p a state such that:

$$w[s \ldots i] : p \bigtriangleup \beta_i \quad 0 \leq s < i \leq n$$

$$\mathsf{Set}(\beta_n) \cap \mathbf{Acc} = \emptyset$$

Then for any state q appearing within some β_i, and for any σ, the following run exists:

$$w \Vdash (q, i)\sigma \overset{*}{\rightsquigarrow} \sigma$$

Proof By induction on $(n - i)$. For the base case ($i = n$), we have the machine:

$$(q, n)\, \sigma$$

Since $q \in \beta_n$, this is not an accepting configuration. Therefore, we have:

$$w \Vdash (q, n)\, \sigma \rightsquigarrow \sigma$$

For the inductive step, suppose $i = k$, $(s < k < n)$, then we have the machine:

$$(q, k)\, \sigma$$

If q has no transitions on $w[k]$, the proof is trivial. Let us assume:

$$w[k] : q \mapsto q'_0 \ldots q'_m$$

Then we have the transition:

$$w \Vdash (q, k)\, \sigma \rightsquigarrow (q'_0, k + 1) \ldots (q'_m, k + 1)\, \sigma$$

Furthermore, the definition of $\bigtriangleup$ implies that $q'_0, \ldots, q'_m$ are part of β_{k+1}. Therefore, we can apply the induction hypothesis to each of the newly spawned frames in succession, which leads to the desired result. $\square$

Lemma 5.1.10 Let w be an input string of length n, s a constant offset

into w and p a state such that:

$$w[s \ldots i] : p \triangle \beta_i \quad 0 \leq s < i \leq n$$

$$\mathsf{Set}(\beta_n) \cap \mathbf{Acc} = \emptyset$$

Then for any state q appearing within some β_i, the following run exists (for some σ):

$$w \Vdash (p, s) \overset{*}{\rightsquigarrow} (q, i)\sigma$$

Proof By induction on $(i - s)$. For the base case ($i = s + 1$), suppose we have:

$$w[s] : p \mapsto q_0 q_1 \ldots q_m$$

Which gives us the transition:

$$w \Vdash (p, s) \rightsquigarrow (q_0, s + 1) \ldots (q_m, s + 1)$$

From the definition of $\triangle$, $\beta_{s+1} = q_0 \ldots q_m$. Therefore, applying Lemma 5.1.9 in sucession (to the newly spawned frames) yields the required result.

For the inductive step, suppose $i = k + 1$ ($s < k < n$) and $\dot{q}$ appears in β_{k+1}. Then from the definition of $\triangle$, there must be some q' appearing in β_k such that:

$$\beta_k = \beta\, q'\, \beta' \qquad w[k + 1] : q' \twoheadrightarrow q_0 \ldots \dot{q} \ldots q_m$$

Now from the induction hypothesis we get:

$$w \Vdash (p, s) \overset{*}{\rightsquigarrow} (q', k)\, \sigma$$

Therefore, we deduce the run:

$$w \Vdash (p, s) \overset{*}{\rightsquigarrow} (q', k)\sigma \rightsquigarrow (q_0, k + 1) \ldots (\dot{q}, k + 1) \ldots (q_m, k + 1)\, \sigma$$

As before, applying Lemma 5.1.9 to the newly spawned frames yields the final result. □

Lemma 5.1.11 For any backtracking machine run:

$$w \Vdash \sigma \overset{n}{\rightsquigarrow} \sigma'$$

And for any $\bar{\sigma}$, the following run also exists:

$$w \Vdash \sigma\, \bar{\sigma} \overset{n}{\rightsquigarrow} \sigma'\, \bar{\sigma}$$

Proof Observe that each transition taken by the the first machine can be simulated on the extended machine. Moreover, each transition of the extended machine leaves the additional $\bar{\sigma}$ untouched. □

The backtracking machine performs a leftmost, depth-first traversal of the search tree. We need to consider situations where some machine state σ consists of nodes in the search tree above some horizontal cut β. If the input string is x of length n, that means that for each state/index pair (p_j, i_j) in σ, the substring $x[i_j..n]$ takes us from p_j to some β_j, and together these β_j make up β.

Definition 5.1.12 For a string x of length n, a machine state σ and a NFA state sequence β, the judgement

$$x : \sigma \Downarrow \beta$$

holds under the following conditions: there are a natural number m, natural numbers $i_1, \ldots, i_m\, (\leq n)$, states $p_1, \ldots p_m$, state sequences $\beta_1, \ldots, \beta_m$ such

that:

$$\sigma = (p_1, i_1) \dots (p_m, i_m)$$

$$\beta = \beta_1 \dots \beta_m$$

$$x[i_j..n] : p_j \triangle \beta_j \qquad (1 \leq j \leq m)$$

Lemma 5.1.13 (Tree traversal) Let w, z be input strings, σ a machine state and β, β' multi-states such that:

$$w : \sigma \Downarrow \beta$$

$$z : \beta \triangle \beta'$$

$$\mathsf{Set}(\beta') \cap \mathbf{Acc} = \emptyset$$

Then for any $\bar{\sigma}$ there exists a machine run:

$$wz \Vdash \sigma\, \bar{\sigma} \overset{n}{\rightsquigarrow} \bar{\sigma}$$

Where $n \geq |\beta|$.

Proof By induction on $|\sigma|$. For the base case, suppose $\sigma = (p, s)$ where $0 \leq s < |w|$. Then from Lemma 5.1.9 we deduce the run:

$$wz \Vdash \sigma \overset{*}{\rightsquigarrow} [\,]$$

That is, any intermediate frame spawned by (p, s) is going to be rejected eventually. Moreover, Lemma 5.1.10 implies that this run visits all the states of β. Finally, Lemma 5.1.11 allows this run to be extended with any failure continuation $\bar{\sigma}$, giving:

$$wz \Vdash \sigma\, \bar{\sigma} \overset{n}{\rightsquigarrow} \bar{\sigma}$$

Where $n \geq |\beta|$. For the inductive step, consider the following inference:

$$\frac{w : (p, s) \Downarrow \beta \qquad\qquad w : \sigma' \Downarrow \beta'}{w : (p, s)\sigma' \Downarrow \beta\beta'}$$

From this we deduce the run:

$$wz \Vdash (p, s)\sigma'\bar{\sigma} \overset{n_1}{\rightsquigarrow} \sigma'\bar{\sigma} \overset{n_2}{\rightsquigarrow} \bar{\sigma}$$

Where $n_1 \geq |\beta|$ (base case) and $n_2 \geq |\beta'|$ (I.H). □

In sum, we have shown that the pumped part of the search tree grows exponentially in the size of the input, and that the backtracking machine is forced to traverse all of it.

Theorem 5.1.14 (Redos analyis soundness) Let the strings x, y and z be constructed by the REDoS analysis. Let k be an integer. Then the backtracking machine takes at least 2^k steps on the input string $x\, y^k\, z$

Proof Follows from Lemma 5.1.8 and Lemma 5.1.13. □

5.2 Completeness of the analysis

The analysis assumes that only a pumpable NFA can lead to an exponential runtime vulnerability. For completeness, we need to ensure that there are no other configurations that can cause such a vulnerability. Here we show that for any non-pumpable NFA, the width of any search tree is bounded from above by a polynomial. In places where an NFA is involved in a discussion below, a non-pumpable NFA is to be assumed (unless otherwise mentioned).

Definition 5.2.1 For an ordered multi-state β and a state p, we define the

function $[\beta]_p$ as follows:

$$[\beta]_p = \begin{cases} 1 + [\beta']_p & \text{if } \beta = p\beta' \\ [\beta']_p & \text{if } \beta = q\beta' \wedge q \neq p \end{cases}$$

Definition 5.2.2 We define the relation $\overset{p}{\sim}$ on ordered multi-states as follows:

$$\beta \overset{p}{\sim} \beta' \Leftrightarrow [\beta]_p = [\beta']_p$$

It can be shown that $\overset{p}{\sim}$ is reflexive, symmetric and transitive.

Definition 5.2.3 The relation $\simeq$ is defined on ordered multi-states as follows:

$$\beta \simeq \beta' \Leftrightarrow \forall\, p \in Q \;.\; \beta \overset{p}{\sim} \beta'$$

It can be shown that $\simeq$ is reflexive, symmetric and transitive.

Lemma 5.2.4 The following properties hold with respect to $\simeq$:

$$\beta \simeq \beta' \Rightarrow \mathsf{Set}(\beta) = \mathsf{Set}(\beta') \wedge |\beta| = |\beta'|$$

$$\beta_1\beta_2 \simeq \beta_3\beta_4 \Leftrightarrow \forall \beta \;.\; \beta_1\beta\beta_2 \simeq \beta_3\beta\beta_4$$

$$\beta \simeq \beta_1\beta'\beta_2 \wedge \beta' \simeq \beta'' \Rightarrow \beta \simeq \beta_1\beta''\beta_2$$

$$\beta_1 \simeq \beta_1' \wedge \beta_2 \simeq \beta_2' \Rightarrow \beta_1\beta_2 \simeq \beta_1'\beta_2'$$

Lemma 5.2.5 Let w be an input string, β_1, β_2 be ordered multi-states such that:

$$\beta_1 \simeq \beta_2 \quad w : \beta_1 \triangle \beta_1' \quad w : \beta_2 \triangle \beta_2'$$

Then $\beta_1' \simeq \beta_2'$.

Proof The base case ($w = \varepsilon$) holds from the definition of $\triangle$. For the inductive step, suppose:

$$w : \beta_1 \triangle p_1 \ldots p_n \quad (\bar{\beta}_1) \qquad a : p_i \mapsto \theta_i \quad (1 \leq i \leq n)$$

$$w : \beta_2 \triangle q_1 \ldots q_n \quad (\bar{\beta}_2) \qquad a : q_j \mapsto \theta'_j \quad (1 \leq j \leq n)$$

So that we have (from the definition of $\triangle$):

$$\beta'_1 = \theta_1 \ldots \theta_n \qquad \beta'_2 = \theta'_1 \ldots \theta'_n$$

Now we perform an inner induction over n. The base case holds trivially as the projection of two equal states ($p_1 = q_1$ from the outer I.H) on the same input character (a) is the same (so $\theta_1 = \theta'_1$). For the inductive step, note that the two new state introduced to $\bar{\beta}_1$ and $\bar{\beta}_2$ (to make them $n+1$ in size) must be the same (again from the outer I.H), which in turn ensures that the relation $\beta'_1 \simeq \beta'_2$ is preserved. $\square$

Definition 5.2.6 Given an NFA, a path γ is a sequence of triples

$$(p_0, a_0, q_0) \ldots (p_n, a_n, q_n)$$

where for $1 \leq i < n$ two successive triples are compatible in the sense that $p_{i+1} = q_i$ and for each triple there is a transition $a : p_i \twoheadrightarrow q_i$ in the NFA. We write $\mathrm{dom}(\gamma)$ for the first node p_0 and $\mathrm{cod}(\gamma)$ for the last node q_n in the path. The sequence of input symbols $a_0 \ldots a_n$ along the path is written as $\overline{\gamma}$ and called the label of the path.

Definition 5.2.7 Let γ be a path:

$$(p_0, a_0, p_1) \ldots (p_{n-1}, a_{n-1}, p_n)$$

The set of nodes $\{p_0, \ldots, p_n\}$ along γ is written as $\mathsf{nodes}(\gamma)$.

Lemma 5.2.8 Given a tree judgement:

$$w : p \triangle \beta$$

For any state q such that $\beta = \beta_1 q \beta_2$, there exists a path γ with:

$$\mathrm{dom}(\gamma) = p \quad \mathrm{cod}(\gamma) = q \quad \overline{\gamma} = w$$

Definition 5.2.9 We write

$$w : p \rightrightarrows q$$

for $\exists p_1, p_2, w_1, w_2$ such that

$$p_1 \neq p_2 \quad w = w_1\, w_2$$

$$w_1 : p \twoheadrightarrow p_1 \quad w_1 : p \twoheadrightarrow p_2$$

$$w_2 : p_1 \twoheadrightarrow q \quad w_2 : p_2 \twoheadrightarrow q$$

5.2.1 Polynomial bound

Definition 5.2.10 Let γ be a path. We define the sets $\mathcal{S}(\gamma)$ and $\mathcal{F}(\gamma)$ as follows:

$$\begin{aligned}\mathcal{S}(\gamma) = \{p \mid \exists\, \gamma_1, \gamma_2 \,.\, \gamma = \gamma_1\gamma_2 \\ \wedge\, \overline{\gamma}_2 : \mathrm{dom}(\gamma_2) \rightrightarrows \mathrm{cod}(\gamma_2) \wedge p = \mathrm{dom}(\gamma_2)\}\end{aligned}$$

$$\mathcal{F}(\gamma) = Q \setminus \mathcal{S}(\gamma)$$

Lemma 5.2.11 Suppose γ is a path such that $p = \mathrm{cod}(\gamma)$. Then the following holds for any w:

$$w : p \triangle \beta \Rightarrow \mathsf{Set}(\beta) \subseteq \mathcal{F}(\gamma)$$

Proof From the definitions we have:

$$\mathsf{Set}(\beta) \subseteq Q = \mathcal{S}(\gamma) \cup \mathcal{F}(\gamma)$$

Suppose $q \in \mathsf{Set}(\beta) \cap \mathcal{S}(\gamma)$. Then $q \in \mathcal{S}(\gamma)$ gives:

$$\exists\, w' \,.\, w' : q \rightrightarrows p$$

However, since $q \in \mathsf{Set}(\beta)$ we also have:

$$w : p \twoheadrightarrow q$$

Leading to the contradiction:

$$w'w : q \rightrightarrows p \twoheadrightarrow q$$

Therefore, it must be the case that $\mathsf{Set}(\beta) \cap \mathcal{S}(\gamma) = \emptyset$. This leads to the conclusion:

$$\mathsf{Set}(\beta) \subseteq \mathcal{F}(\gamma)$$

□

Lemma 5.2.11 is illustrated in Figure 5.3. Note that the fringes of the sibling trees rooted at the two p's are identical ($\triangle$ logic is deterministic), making it impossible for either of them to contain a q (q would be pumpable otherwise).

Definition 5.2.12 We define the reduction $\triangleright$ on pairs of ordered multi-states according to the following rules:

$$(q_1 \ldots q_n, \beta_1 q \beta_2) \triangleright (q_1 \ldots q_i q \ldots q_n, \beta_1 \beta_2) \quad (\exists\, i \,.\, q = q_i)$$

$$(q_1 \ldots q_n, \beta_1 q \beta_2 q \beta_3) \triangleright (q_1 \ldots q_n q q, \beta_1 \beta_2 \beta_3) \quad (\forall\, i \,.\, q \neq q_i)$$

The reduction $\triangleright$ repeatedly groups recurring states. Given that each transition decreases the length of the second component, the reduction must

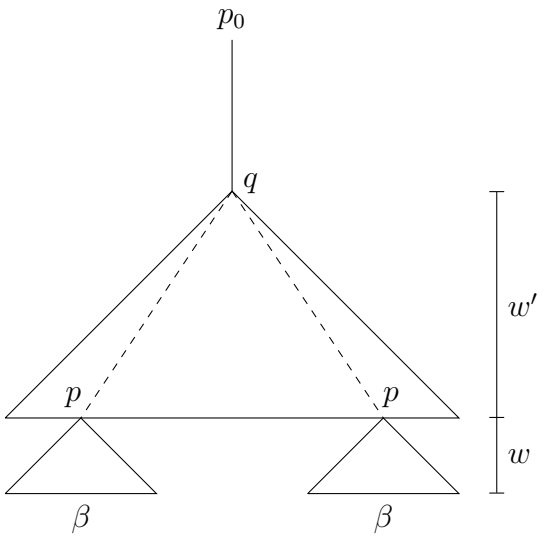


Figure 5.3: Sibling restriction on $\mathcal{S}(\gamma)$

terminate. We use the notation $\triangleright\!\!\triangleright$ to denote a maximal reduction:

$$(\alpha_1, \beta_1) \triangleright \triangleright (\alpha_2, \beta_2) \Rightarrow \nexists (\alpha_3, \beta_3) \;.\; (\alpha_2, \beta_2) \triangleright (\alpha_3, \beta_3)$$

Lemma 5.2.13 Suppose:

$$(\varepsilon, \beta) \triangleright \triangleright (\alpha, \sigma)$$

Then the following properties hold:

$$\beta \simeq \alpha\sigma \tag{a}$$

$$\mathsf{Set}(\alpha) \cup \mathsf{Set}(\sigma) = \mathsf{Set}(\beta) \tag{b}$$

$$\forall\, p \in \mathsf{Set}(\alpha) \;.\; [\alpha]_p = [\beta]_p > 1 \tag{c}$$

$$|\sigma| = |\mathsf{Set}(\sigma)| \tag{d}$$

Definition 5.2.14 We define the semantics:

$$w : (\beta, \alpha, \sigma) \not\trianglelefteq (\beta', \alpha', \sigma')$$

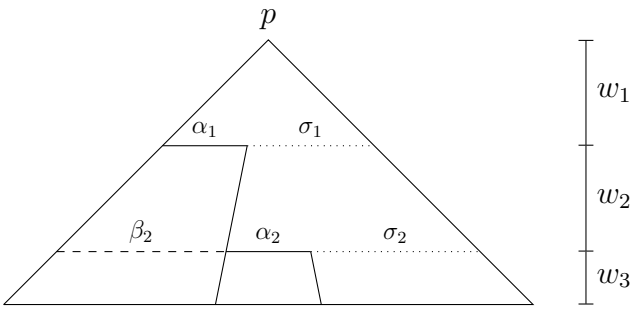


Figure 5.4: An example $\blacktriangle\!\!\!\blacktriangle$ derivation.

on search tree logic with the following inference rules:

$$\frac{a : \beta_1\alpha_1 \triangle \beta_2 \quad a : \sigma_1 \triangle \beta_3 \quad (\varepsilon, \beta_3) \triangleright\triangleright (\alpha_2, \sigma_2)}{a : (\beta_1, \alpha_1, \sigma_1) \blacktriangle\!\!\!\blacktriangle (\beta_2, \alpha_2, \sigma_2)}$$

$$\frac{w : (\beta_1, \alpha_1, \sigma_1) \blacktriangle\!\!\!\blacktriangle (\beta_2, \alpha_2, \sigma_2) \qquad a : (\beta_2, \alpha_2, \sigma_2) \blacktriangle\!\!\!\blacktriangle (\beta_3, \alpha_3, \sigma_3)}{wa : (\beta_1, \alpha_1, \sigma_1) \blacktriangle\!\!\!\blacktriangle (\beta_3, \alpha_3, \sigma_3)}$$

The $\blacktriangle\!\!\!\blacktriangle$ semantics recursively re-arranges the search tree into β, α and σ components at each depth. A derivation using the $\blacktriangle\!\!\!\blacktriangle$ semantics may be visualized as in Figure 5.4. Note that in this hypothetical derivation, we encounter repeated states at depth w_1, thus giving rise to the first non-empty α component (α_1). From w_1 to w_1w_2, we have non-empty β and σ components. Again at depth w_1w_2 we can observe a non-empty α component, which is the result of the previous σ component generating duplicates at this depth. The β component can be thought of as the shadow/projection of all the previous α components.

Lemma 5.2.15 Let p be a state and w an input string such that:

$$w : p \triangle \beta \qquad w : (\varepsilon, \varepsilon, p) \blacktriangle\!\!\!\blacktriangle (\beta', \alpha, \sigma)$$

Then $\beta'\alpha\sigma \simeq \beta$.

Proof By induction on the length of w. For the base case ($w = a$), suppose:

$$a : w \triangle \beta$$

Then from the definition of $\trianglelefteq$ we get:

$$a : (\varepsilon, \varepsilon, p) \trianglelefteq (\varepsilon, \alpha, \sigma)$$

Where $(\varepsilon, \beta) \triangleright\triangleright (\alpha, \sigma)$. Therefore, Lemma 5.2.13 (a) gives: $\beta \simeq \alpha\sigma$.

For the inductive step ($w = w'a$), suppose:

$$w' : p \triangle \beta_1 \tag{A.1}$$

$$w' : (\varepsilon, \varepsilon, p) \trianglelefteq (\beta_1', \alpha_1, \sigma_1) \tag{A.2}$$

Then the induction hypothesis yields: $\beta_1 \simeq \beta_1'\alpha_1\sigma_1$ (I.H). Now let us assume:

$$a : \beta_1 \triangle \beta_2 \tag{B.1}$$

$$a : \beta_1'\alpha_1 \triangle \beta_2' \tag{B.2}$$

$$a : \sigma_1 \triangle \beta_{3'} \tag{B.3}$$

$$(\varepsilon, \beta_3') \triangleright\triangleright (\alpha_2, \sigma_2) \tag{B.4}$$

Assumptions (A.1), (B.2) - (B.4) and the definition of $\trianglelefteq$ leads to:

$$w' : (\varepsilon, \varepsilon, p) \trianglelefteq (\beta_2', \alpha_2, \sigma_2)$$

Moreover, assumptions (B.2), (B.3) implies:

$$a : \beta_1'\alpha_1\sigma_1 \triangle \beta_{2'}\beta_{3'}$$

Therefore, Lemma 5.2.5 yields (with I.H, B.1):

$$\beta_2 \simeq \beta_{2'}\beta_{3'}$$

Furthermore, Lemma 5.2.13 (a) implies (with B.4):

$$\beta_{3'} \simeq \alpha_2\sigma_2$$

Finally, Lemma 5.2.4 (c) leads to:

$$\beta_2 \simeq \beta_{2'}\alpha_2\sigma_2$$

□

Lemma 5.2.16 Let γ be a path with $p = \mathrm{cod}(\gamma)$ and w an input string such that:

$$w : (\varepsilon, \varepsilon, p) \text{🕰} (\beta, \alpha, \sigma)$$

Then the following properties hold:

$$\mathsf{Set}(\beta\alpha\sigma) \subseteq \mathcal{F}(\gamma) \quad \text{(a)}$$

$$|\sigma| \leq |\mathcal{F}(\gamma)| \quad \text{(b)}$$

$$|\alpha\sigma| \leq |\mathcal{F}(\gamma)| * o \quad \text{(c)}$$

Proof For property (a), suppose $w : p \triangle \beta'$. Then from Lemma 5.2.11 we get:

$$\mathsf{Set}(\beta') \subseteq \mathcal{F}(\gamma)$$

Moreover, Lemma 5.2.15 gives:

$$\beta\alpha\sigma \simeq \beta'$$

Therefore, the desired result follows from Lemma 5.2.4 (a).

For property (b), note that it follows from property (a) that

$$\mathsf{Set}(\sigma) \subseteq \mathcal{F}(\gamma)$$

The definition of $\trianglelefteq$ yields:

$$\exists\ \beta' \ .\ (\varepsilon, \beta') \triangleright \triangleright (\alpha, \sigma)$$

Therefore, from Lemma 5.2.13 (d) it follows that:

$$|\sigma| = |\mathsf{Set}(\sigma)| \leq |\mathcal{F}(\gamma)|$$

For property (c), suppose $w = w'a$ (the result holds trivially for $w = \varepsilon$). Then from the definition of $\trianglelefteq$ there exist variables σ', β' such that:

$$w' : (\varepsilon, \varepsilon, p) \trianglelefteq (\_, \_, \sigma') \tag{A.1}$$

$$a : \sigma' \triangle \beta' \tag{A.2}$$

$$(\varepsilon, \beta') \triangleright \triangleright (\alpha, \sigma) \tag{A.3}$$

From (A.2) and the structure of the NFA, we derive:

$$|\beta'| \leq |\sigma'| * o$$

Assumption (A.3) and Lemma 5.2.13 (a) implies:

$$\alpha\sigma \simeq \beta'$$

Therefore, Lemma 5.2.4 (a) and property (b) above leads to:

$$|\alpha\sigma| = |\beta'| \leq |\sigma'| * o \leq |\mathcal{F}(\gamma)| * o$$

□

Lemma 5.2.17 Let w be an input string and p a state. Let i, k be indices such that:

$$0 < i < k \leq |w|$$

$$w[0 \ldots i] : (\varepsilon, \varepsilon, p) \trianglelefteq (\beta_i, \alpha_i, \sigma_i)$$

$$w[i \ldots k] : \alpha_i \triangle \alpha_{(i,k)}$$

Then $\beta_k = \alpha_{(1,k)} \ldots \alpha_{(k-1,k)}$

Proof By induction on k. □

With reference to Figure 5.4, Lemma 5.2.17 establishes the connection between the fringe of the overall triangle and those of individual trapezoidal slices.

Lemma 5.2.18 Let γ be a path with $p = \text{cod}(\gamma)$ and w an input string such that:

$$w : (\varepsilon, \varepsilon, p) \trianglelefteq (\beta, \alpha, \sigma)$$

Then for a state q such that $\alpha = \alpha_1 q \alpha_2$ (for some α_1 and α_2), there exists a path γ' from p to q such that:

$$\mathcal{F}(\gamma\gamma') \subset \mathcal{F}(\gamma)$$

Proof Follows from Lemma 5.2.11 (q is repeated inside α). □

Lemma 5.2.19 Suppose γ is a path with $p = \text{cod}(\gamma)$ and w an input string of length n such that:

$$w : p \triangle \beta$$

Then the following holds:

$$|\beta| < k^k * o^k * n^k$$

Where $k = |\mathcal{F}(\gamma)|$.

Proof We perform an induction on k.

Base case - 1: Suppose $k = 0$. Then it follows from Lemma 5.2.11 that $|\beta| = 0$, which is within the bounds of our polynomial.

Base case - 2: Suppose $k = c$ (for some constant c) and:

$$\nexists \gamma' \,.\, \gamma' = \gamma\gamma'' \wedge \mathcal{F}(\gamma') < c$$

This means the search tree rooted at $\mathrm{cod}(\gamma)$ cannot contain duplicates at any depth, for if it does, we can always find an extended path γ' for which $\mathcal{F}(\gamma')$ is less. This restriction immediately implies that the fringe of the search tree cannot grow beyond c, which is well within the bounds of our (over-estimating) polynomial $(c^c * o^c * n^c)$.

Inductive step: Let us assume the notation:

$$w[0 \ldots i] : (\varepsilon, \varepsilon, p) \trianglerighteq (\beta_i, \alpha_i, \sigma_i)$$

$$w[i \ldots n] : \alpha_i \triangle \alpha_{(i,n)}$$

Where $0 < i < n$. From Lemma 5.2.17 we deduce:

$$\beta_n \alpha_n \sigma_n = \alpha_{(1,n)} \ldots \alpha_{(n-1,n)} \alpha_n \sigma_n \tag{A}$$

Note that it follows from Lemma 5.2.18 that we can apply the induction hypothesis to each path ending in some state within an α_i. Therefore, we derive:

$$\forall i \; \exists v < k \,.\, \left|\alpha_{(i,n)}\right| < |\alpha_i| * v^v * o^v * |w[i \ldots n]|^v$$

In terms of the illustration in Figure 5.4, this statement measures the bottom edges of the trapezoids. Now, taking into account that $v < k$ and $|w[i \ldots k]| < n$, we arrive at:

$$\forall i \,.\, \left|\alpha_{(i,n)}\right| < |\alpha_i| * k^k * o^k * n^k$$

Moreover, it follows from Lemma 5.2.16 (c) that:

$$|\alpha_i| \leq k * o$$

Therefore, we get:

$$\forall i \ . \ \left|\alpha_{(i,n)}\right| < k^{k+1} * o^{k+1} * n^k \tag{B}$$

Now, we combine (A) and (B) to obtain:

$$|\beta_n \alpha_n \sigma_n| < (n-1) * k^{k+1} * o^{k+1} * n^k + |\alpha_n \sigma_n|$$

Furthermore, it follows from Lemma 5.2.16 (c) that:

$$|\alpha_n \sigma_n| \leq k * o < k^{k+1} * o^{k+1} * n^k$$

Therefore, we get:

$$|\beta_n \alpha_n \sigma_n| < k^{k+1} * o^{k+1} * n^{k+1}$$

Since we know $\beta \simeq \beta_n \alpha_n \sigma_n$ from Lemma 5.2.15, the inductive step holds. □

Theorem 5.2.20 (Redos analysis completeness) Given an NFA with an exponential runtime vulnerability, the REDoS analysis presented in Section 4.3 will produce an attack string which triggers this behaviour on a backtracking regular expression matcher.

Proof Lemma 5.2.19 implies that for a non-pumpable NFA, the search tree width is polynomially bounded. Since w is finite, the entire search space in turn becomes polynomially bounded. This suggests that only a pumpable NFA can lead to an exponentially large search space. Finally, the analysis presented in Section 4.3 is exhaustive in that if a suitable attack string exists for a pumpable NFA, it will eventually be found. □

CHAPTER 6

RXXR

Having established the correctness of the analysis, we now demonstrate the usefulness of a tool we developed (code-named *RXXR*) that implements the said analysis.

6.1 Implementation

We implemented the analysis presented above in OCaml [TR14]. Apart from the code used for parsing regular expressions (and some other boilerplate code), the main source modules have an almost one-to-one correspondence with the concepts discussed thus far. This relationship is illustrated in Table 6.1.

Each module interface (`.mli` file) contains function definitions which directly correspond to various aspects of the analysis presented earlier. For an example, the NFA module provides the following function for querying ordered transitions:

Concept (Theory)	Implementation (OCaml Module)
NFA	`Nfa.mli/ml`
β	`Beta.mli/ml`
Φ	`Phi.mli/ml`
$\twoheadrightarrow_{\mathbf{2}}$	`Product.mli/ml`
$\twoheadrightarrow_{\mathbf{3}}$	`Triple.mli/ml`
Prefix analysis	`XAnalyser.mli/ml`
Pumpable analysis (y_1)	`Y1Analyser.mli/ml`
Pumpable analysis (ay_2)	`Y2Analyser.mli/ml`
Suffix analysis	`ZAnalyser.mli/ml`
Overall analysis	`AnalyserMain.mli/ml`

Figure 6.1: Theory to source-code correspondence

```
val get_transitions : Nfa.t -> int ->
                      ((char * char) * int) list;;
```

The NFA states are represented as integers. Each symbol of the input alphabet is encoded as a pair of characters, allowing a uniform representation of character classes (`[a-z]`) as well as individual characters (`a = [a-a]`).

The NFA used in the implementation (`Nfa.mli/ml`) contains ε transitions, which were not part of the NFA formalization presented earlier. The reason for this deviation is that having ε transitions allows us to preserve the structure of the regular expression within the NFA representation, which in turn preserves the order of the transitions. The correctness of the implementation is unaffected given that the ε-NFA translates into an equally expressive ordered NFA. Only a slight mental adjustment (from ordered NFAs to ε-NFAs) is required to correlate the theoretical formalizations to the OCaml code. For an example, Figure 6.2 presents the module interface for β. The function `advance()` is utilized inside the `XAnalyser.ml` module to perform

```
(* internal representation of beta *)
type t;;

module BetaSet : (Set.S with type elt = t);;

(* beta with just one state *)
val make : int -> t;;

(* returns the set of states contained within this beta *)
val elems : t -> IntSet.t;;

(* calculate all one-character reachable betas *)
val advance : (Nfa.t * Word.t * t) -> (Word.t * t) list;;

(* consume all epsilon transitions while recording pumpable
   kleene encounters *)
val evolve : (Nfa.t * Word.t * t) -> IntSet.t ->
               Flags.t * t * (int * t) list;;
```

Figure 6.2: `Beta.mli`

the closure computation (i.e. compute all βs reachable from the root node), whereas `evolve()` is a utility function used to work around the ε transitions. The modules `(Phi / Product / Triple).mli` define similar interfaces for Φ, $\twoheadrightarrow_{\mathbf{2}}$ and $\twoheadrightarrow_{\mathbf{3}}$ constructs introduced in the analysis.

The different phases of the analysis are implemented inside the corresponding analyser modules. As an example, Figure 6.3 presents the `Y2Analyser.mli` module responsible for carrying out the analysis after the branch point ($\twoheadrightarrow_{\mathbf{3}}$ simulation). The internal representation of the analyser (`type t`) holds the state of the closure computation, which is initialized with an initial triple argument through the `init()` function. We defer the interested reader to module definition (`.ml`) files for further details on the implementation.

```
(* internal representation of the analyser *)
type t;;

(* initialize analyser instance for the specified triple and
   the kleene state *)
val init : (Nfa.t * Word.t * Triple.t) -> int -> t;;

(* calculate the next (y2, phi) *)
val next : t -> (Word.t * Phi.t) option;;

(* read analyser flags *)
val flags : t -> Flags.t;;
```

Figure 6.3: `Y2Analyser.mli`

6.2 Evaluation data

The analysis was tested on two corpora of regexes. The first of these was extracted from an online regex library called *RegExLib* [Reg12], which is a community-maintained regex archive; programmers from various disciplines submit their solutions to various pattern matching tasks, so that other developers can reuse these expressions for their own pattern matching needs. The second corpus was extracted from the popular intrusion detection and prevention system *Snort* [Sou12], which contains regex-based pattern matching rules for inspecting IP packets across network boundaries. The contrasting purposes of these two corpora (one used for casual pattern matching tasks and the other used in a security critical application) allow us to get a better view of the seriousness of exponential vulnerabilities in practical regular expressions.

The regex archive for RegExLib was only available through the corresponding website [Reg12]. Therefore, as the first step the expressions had

to be scraped from their web source and adapted so that they can be fed into our tool. These adaptations include removing unnecessary white-space, comments and spurious line breaks. A detailed description of these adjustments as well as copies of both adjusted and un-adjusted data sets have been included with the resources linked from the RXXR distribution [TR14] (also including the Python script used for scraping). The regexes for Snort, on the other hand, are embedded within plain text files that define the Snort rule set. A Python script (also linked from the RXXR webpage) allowed the extraction of these regexes, and no further processing was necessary.

6.3 Results

The results of running the analysis on these two corpora of regexes are presented in Table 6.3. The figures show that we can process around 75% of each of the corpora with the current level of syntax support. Out of these analyzable amounts, it is notable that regular expressions from the RegExLib archive use the Kleene operator more frequently (about 50% of the analyzable expressions) than those from the Snort rule set (close to 30%). About 11.5% of the Kleene-based RegExLib expressions were found to have a pumpable Kleene expression as well as a suitable suffix, whereas for Snort this figure stands around 0.55%.

The tool makes every attempt to analyse a given pattern, even the ones which contain non-regular constructs like backreferences. An expression $(e_1|e_2)$ may be vulnerable due to a pumpable Kleene that occurs within e_1, whereas e_2 might contain a backreference. In these situations, the analyser

	RegExLib	Snort
Total patterns	2992	12499
Parsable	2290	9801
Pumpable	159	19
Vulnerable	131	15
Interrupted	4	0
Pruned	0	2
Time	61.51 (s)	30.10 (s)

Figure 6.4: RXXR2 results - statistics

attempts to derive an attack string which avoids the non-regular construct. If such a non-regular construct cannot be avoided, the analysis is terminated with the `interrupted` flag.

On certain rare occasions, search pruning is employed as an optimization. It is activated when there have been a number of unstable derivations (failing to meet $\Phi_{y2} \subseteq \Phi_x$) for a given prefix. For an example, consider the regular expression:

$$([\hat{}a]^*b)^*[\hat{}c]\{1000\}$$

Here the Kleene expression $([\hat{}a]^*b)^*$ is pumpable for any string which contains two copies of b (e.g. $bb, bab, abb, cbb \ldots$). However, if the analysis were to pick a pumpable string that *does not contain the symbol* c, it will lead to an unstable derivation. Intuitively, the followup expression $[\hat{}c]\{1000\}$ (which has a large state space) will also consume the pumpable string and introduce a new state in Φ_{y2}, breaking the inclusion $\Phi_{y2} \subseteq \Phi_x$. Pruning allows the analysis to attempt different variants of the pumpable string without getting stuck on a single search path where all of the pumpable strings lead to unstable (but unique) derivations (e.g. $bb, bab, baab, baaab, \ldots$). Needless to say, this is an

ad-hoc optimization that can be further improved with more sophisticated heuristics. Given that pruning was only triggered in two instances for the entire data set above, we believe the current heuristic (a static bound on the number of unstable derivations) is adequate. If a pruned search does not report a vulnerability, it should be re-run with a higher (or infinite) prune limit in order to obtain a conclusive result.

6.3.1 Validation

The task of validating vulnerabilities is complicated by the fact that different regular expression implementations (Java, Python, .NET etc.) have different syntax flavours. RXXR itself is written to accept PCRE like patterns of the form `/<REGEX>/<FLAGS>` where `REGEX` contains the main expression and `FLAGS` are used to control various aspects of the matching process (e.g. whether to match multi-line input or not). Java, Python and .NET use separate library calls to configure such behavior. Moreover, they can also differ from one another in terms of the syntax allowed within the main expression. For an example, Java requires tricky escape sequences when working with meta-characters (e.g a literal backslash requires `\\\\`), whereas Python is more flexible with its support for raw (un-interpreted) input strings.

For these reasons we chose Python as our main validation platform (Python's support for raw strings makes the porting relatively simple). A sample of vulnerabilities were then manually validated on other platforms (Java, .NET and PCRE). Following table illustrates how Python responds to above vulnerabilities:

	RegExLib	Snort
Total vulnerabilities	131	15
Successfully validated	115	14
Python parsing bug	12	0
Python not vulnerable	4	1

Figure 6.5: Validation of vulnerabilities - Python

The Python scripts developed for this validation are also included with the RXXR distribution [TR14], along with instructions on how to reproduce the above results. We discovered that Python was not able to compile regular expressions of the form $([a-z]^*)^*$, which is a known Python defect [Tra08]. Variants of this bug affected 12 of the RegExLib vulnerabilities which we could not validate on Python. The remaining few cases were down to trivial vulnerabilities that Python manages to work around. We observed that both Python and .NET are capable of avoiding vulnerabilities in expressions like $([a-c]|b)^*d$ or $(a|a)^*b$, where the redundancies are quite obvious. Interestingly however, Java does not seem to implement any such workarounds; even when matching the expression $(a|a)^*b$ against the input string $a^n (n \sim 50)$, the JVM (Java Virtual Machine) becomes non-responsive.

6.3.2 Sample vulnerabilities

The vulnerabilities reported range from trivial programming errors to more complicated cases. For an example, the following regular expression is meant to validate time values in 24-hour format (from RegExLib):

```
^(([01][0-9]|[012][0-3]):([0-5][0-9]))*$
```

Here the author has mistakenly used the Kleene operator instead of the `?` operator to suggest the presence or non-presence of the value. This pattern works perfectly for all intended inputs. However, our analysis reports that this expression is vulnerable with the pumpable string "`13:59`" and the suffix "`/`". This result gives the programmer a warning that the regular expression presents a DoS security risk if exposed to user-malleable input strings to match.

For a moderately complicated example, consider the following regular expression (again from RegExLib):

```
^([a-zA-z]:((\\([-*\.*\w+\s+\d+]+)|(\w+)\\)+)(\w+.zip)|(\w+.ZIP))$
```

This expression is meant to validate file paths to zip archives. Our tool identifies this expression as vulnerable and generates the prefix "`z:\ `", the pumpable string "`\zzz\`" and the empty string as the suffix. This is probably an unexpected input in the author's eye, and this is another way in which our tool can be useful in that it can point out potential mis-interpretations which may have materialized as vulnerabilities.

Out of the over 12,000 patterns examined, there were two cases that failed to terminate within any reasonable amount of time. Closer inspection reveals that a pumpable Kleene expression with a vast number of states is to blame. Consider the following example (from RegExLib):

```
^(([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.
  ([a-zA-Z]{2,5}){1,25})+
  ([;.](([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.
  ([a-zA-Z]{2,5}){1,25})+)*$
```

If we change the counted expressions of the form `e{1,25}` into `e{1,5}`, the analyser returns immediately. This shows that the analysis itself can take a long time on certain inputs. However, such cases are extremely rare.

6.4 Comparison to fuzzers

REDoS analysers commonly used in practice are based on a brute-force approach known as fuzzing, where the runtime of a pattern is tested against a set of strings. A leading example of this approach is the Microsoft's SDL Regex Fuzzer [Mic11].

As is common with most brute-force approaches, the main problem with fuzzing is that it can take a considerable amount of time to detect a vulnerability. This is especially pronounced in the case of REDoS analysis as vulnerable patterns tend to take increasing amounts of time with each iteration of testing. This property alone disqualifies fuzzing based REDoS analysers from being integrated into code-analysis tools, as their operation would impose unacceptable delays. For an example, consider the following simple pattern:

```
^(a|b|ab)*c$
```

Even with a lenient fuzzer configuration (ASCII only, 100 fuzzing iterations), SDL fuzzer takes 5-10 minutes to report a vulnerability on this pattern. By comparison, our analyser can process tens of thousands of patterns in less time.

Fuzzers can also miss out on vulnerabilities. For an example, consider the following two patterns:

```
^.*|(a|b|ab)*c$
^(a|b|ab)*c|.*$
```

SDL Fuzzer reports both of these patterns as being safe. However, the non-commutative property of the alternation renders the second pattern vulnerable (as explained in Section 4.4). Another such example is:

```
^(a|b|c|ab|bc)*a.*$
```

For this pattern, only one of the pumpable strings (*bc*) can lead to an attack string, and it must not end in an *a*. Such relationships are difficult to be caught in a heuristics-based fuzzer.

Yet another problem with fuzzers is caused by the element of randomness present in their string generating algorithms. Since fuzzers are not based on any sound theory, some form of randomness is necessary in order to increase the chance of stumbling upon a valid attack string. However, this can make the fuzzer yield inconsistent results for the same pattern. Consider the following pattern for an example:

```
(a|b)*[^c].*|(c)*(a|b|ab)*d
```

The SDL fuzzer reports this pattern as being safe in most invocations, but in few cases it finds an attack string.

Finally, the ultimate purpose of using a static analyser is to detect potential vulnerabilities upfront and lead to the corresponding fixes. Our analyser pin-points the exact pumpable Kleene expression and generates a string (pumpable string) which witnesses vulnerability, making the fixing of the error a straightforward task. This is notably in contrast to the fuzzer, which

outputs a random string (mostly in hex format) that does not provide any insight into the source of the problem.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This research was motivated by the observation that practical implementations of regular expressions greatly differ from their theoretical formalizations. Most pattern matching libraries in practice (JAVA, .NET, Python, Perl, Ruby, PCRE etc.) employ syntax-directed backtracking for matching regular expressions. This is a departure from programming language theory / compiler construction, where regular expressions used for lexing are implemented as deterministic finite automata.

Once it was understood that pattern matching in practice requires more expressive matching constructs than those available in pure regular languages, we set out to develop a new semantics framework to formalize these backtracking pattern matchers. Since backtracking resembles a form of continuation passing, abstract machine models seemed a good fit for describing their operational behavior. We aimed to develop an extensible form of semantics which can be used to reason about various matching constructs available in practical pattern matchers. More specifically, we wanted to address an infamous non-termination problem with naive backtracking implementations and better understand the problem of exponential blowups also common with

backtracking pattern matchers.

7.1 Outcome

In Chapter 3 we developed our semantics for regular expressions starting from a coarse-grained big-step semantics and progressively refining it to obtain more concrete abstract machine models. We demonstrated the versatility of our approach by developing separate machine models for the lock-step pattern matcher as well as the backtracking pattern matcher. We believe we successfully achieved one of the critical goals of our research in that using these machine models we were able to establish the correctness of both the backtracking matcher (Section 3.2) as well as the lock-step matcher (Section 3.3), the termination proof of the EKWF machine in particular is quite novel and much simpler in comparison to the existing results in literature (discussed below under related work).

The culmination of this research however is the development of a sound and complete static analysis for exponential blowups in backtracking pattern matchers (Chapters 4, 5 and 6). This contribution is quite significant given that prior to this work, exponential blowups were only understood as an inherent property of the backtracking paradigm [Cox07, Sul13, OWA12], without a definitive explanation of the syntactic configurations which cause them or how they can be analysed for. Perhaps the most convincing argument in favor of this claim is that the best defenses against exponential blowups prior to this work were limited to visual debugging [Goy09b] and fuzzing techniques [Mic11], which we have shown to compare very poorly

against our analyser implementation (Chapter 6).

Finally, our work has also sparked interest among other researchers on general-purpose backtracking pattern matching. The work presented in [BDvdM14] has been inspired by one of our publications detailing an earlier version of our static analysis [KRT13]. This earlier analyser was unsound. We spent a significant portion of effort towards repairing its correctness, the completeness proof was especially demanding. We hope the inclusion of this result and the demonstrated effectiveness of the practical implementation would generate further interest in the subject. We have also been cited in [SM14], where the authors present a static analysis for detecting linear runtimes. Such an analysis would be useful for applications that need to make specific guarantees on runtime behavior. As discussed below, we believe that both of these analyses can be collapsed into one by making it possible to detect both exponential vulnerabilities as well as polynomial complexities.

7.2 Related work

The literature review (Chapter 2) covered the broad spectrum of background research relevant to the present work. In the following sections we reflect on some of those literature in the light of the work presented here. We also discuss a few additional literature that have noteworthy links to the theories developed in the present thesis.

7.2.1 Machines

The EKW machine focuses in on a current expression while maintaining a continuation for keeping track of what to do next. In that sense, the machine is a distant relative of machines for interpreting lambda terms, such as the SECD machine [Lan64] or the CEK machine [FF86]. On the other hand, regular expressions are a much simpler language to interpret than lambda calculus, so that continuations can be represented by a single pointer into the tree structure (or to machine code in Thompson's original implementation). While the idea of continuations as code pointers is sometimes advanced as a helpful intuition, the representation of continuations in CPS compiling [App92] is more complex, involving an environment pointer as well. To represent pointers and the structures they build up, we found it convenient to use a small fragment of separation logic [Rey02], given by just the separating conjunction and the points-to-predicate. (They are written as $\otimes$ and $\pi(p) = e$ in Chapter 3, to avoid clashes with other notation.)

Backtracking is a classic application of continuations, and regular expression matchers similar to the backtracking machine have been investigated in the functional programming literature [DN01, Har99, FC04]. Other recent work on regular expressions in the programming language community includes regular expression inclusion [HN11] and submatching [SL12].

The earliest known presentation of the issue of non-termination in backtracking pattern matchers is given in [Har99]. The author suggests rewriting (pre-processing) the input expression into an equivalent normal form that avoids the possibility of infinite loops. This technique would only work for

purely-regular expressions, whereas the EKW/EKWF machines (and the termination proof) can be easily extended to support non-regular constructs. In fact, one of the main advantages of backtracking pattern matchers is that they are flexible enough to allow non-regular extensions in the pattern specification language [Cox07]. Also, as noted in [FC04], the rewriting technique suggested in [Har99] can explode the size of the input expression. The EKWF machine corresponds to the first-order pattern matcher presented in [DN01]. The matcher presented there is written in ML, whereas the EKWF machine is independent of any implementation language. The accompanying correctness proof depends on an intricate ordering imposed on the sizes of three predicates, whereas our termination proof involves a relatively simpler inductive argument.

7.2.2 Analysis

Program analysis for security is by now a well established field [CM04]. REDoS is known in the literature as a special case of algorithmic complexity attacks [CW03, SEJ06].

Apart from some basic constructions like the power DFA covered in standard textbooks [HU79], we have not explicitly relied on automata theory. Instead, we regarded the matcher as an abstract machine that can be analyzed with tools from programming language research. Specifically, the techniques in this work are inspired by substructural logics, such as Linear Logic [Gir87, GLT89] and Separation Logic [IO01, Rey02]. Concerning the latter, it may be instructive to compare the sharing of w or absence of shar-

ing of β in Figure 5.1 to the connective of Separation logic. In a conjunction, the heap h is shared:

$$\frac{h \models P_1 \qquad h \models P_2}{h \models P_1 \wedge P_2}$$

By contrast, in a separating conjunction, the heap is split into disjoint parts that are not shared:

$$\frac{h_1 \models P_1 \qquad h_2 \models P_2 \qquad h_1 \cap h_2 = \emptyset}{h_1 \cup h_2 \models P_1 * P_2}$$

Tree-shaped data structures have been one of the leading examples of separation logic. However, a difference to the search trees we have used in this thesis is that the whole search tree is not actually constructed as a data structure in memory. Rather, only a diagonal cut across it is maintained at any time in the backtracking machine. The whole tree does not exist in memory, but only in space *and* time, so to speak. In that regard the search trees are like parse trees, which the parser only needs to construct in principle by traversing them, and not necessarily as a data structure in memory complete with details of all nodes [AP97, ALSU07].

7.3 Limitations and future work

We now turn towards those topics that could not be addressed within the time-frame of this thesis. We also discuss some limitations of the present research and identify potential future directions of work in the wider domain of string pattern matching.

7.3.1 Machines

With regards to the EKWF machine, one of the results missing from this thesis is the total correctness; the correctness of the machine needs to be re-established after the introduction of the barriers. While it is trivial to prove that the results calculated by the EKWF machine (with barriers) are correct, we also need to show that the barriers only terminate infinite loops - not valid runs of the machines. This is a reasonable result to believe, but a formal proof of the fact remains to be developed.

One of the motivations for this thesis was to develop a semantics framework that would be easily extensible to non-regular pattern constructs. The work on the machines and the analysis however kept us from exploring this territory. We now briefly discuss how this goal can be approached.

Submatch extraction [Lau01] is a common pattern extension implemented by most regular expression libraries, let use denote such expressions with the syntax $(\mathbf{x} : e)$. That is, variable $\mathbf{x}$ should contain the substring matched by e at the end of a matching operation. We can augment the basic EKWF machine to capture this operation with the following transitions:

$$M \vdash \langle (\mathbf{x} : e) \; ; \; k \; ; \; w \rangle :: f \rightarrow M[\mathbf{x} \rightarrow (w, \_)] \vdash \langle e \; ; \; \bar{\mathbf{x}} :: k \; ; \; w \rangle :: f$$

$$M \vdash \langle \bar{\mathbf{x}} \; ; \; k \; ; \; w \rangle :: f \rightarrow M[\mathbf{x} \rightarrow (w', w)] \vdash \langle e \; ; \; k \; ; \; w \rangle :: f$$

Here the heap M contains the submatch mappings encountered throughout the matching process, $\bar{\mathbf{x}}$ is a special tag expression used to denote the end of a marked expression $(\mathbf{x} : e)$.

Closely associated with submatching is the topic of quantifiers. Given the

pattern $(\mathbf{x} : (ab)^*)b^*$, an input string of the form *abababbb* can be matched in more than one way; whether $\mathbf{x}$ will consume the entire string or if it would only match the prefix *ababa* is determined by the corresponding quantifier (greedy, reluctant etc.) in force. Understanding the operational behavior of such constructs and the different matching standards (POSIX, PCRE etc. [Cox10, Sul12]) would be an interesting future research direction.

Finally, backreferences represent one of the most powerful irregular pattern constructs [Aho90]. Denoting a backreference with $\backslash\mathbf{x}$, we can further extend our EKWF machine to support this construct in the following manner:

$$M \vdash \langle \backslash\mathbf{x} \; ; \; k \; ; \; w \rangle :: f \rightarrow M \vdash \langle r(M[\mathbf{x}]) \; ; \; k \; ; \; w \rangle :: f$$

Where the function $r(M[\mathbf{x}])$ constructs a literal regular expression from the substring mapping $M[\mathbf{x}]$. Note that we have left out the problem of invalid backreferences here (i.e. $\mathbf{x}$ not present in store M upon reaching $\backslash\mathbf{x}$), which is an interesting problem on its own. In any case, the versatility of the EKWF machine is quite evident from these examples. As future work, we hope to further develop these abstract machine models to enable reasoning about various irregular pattern constructs [Cox11] implemented by practical pattern matching libraries.

7.3.2 Analysis

At present, the analysis constructs attack strings whenever there is an exponential runtime vulnerability. An interesting extension would be to compute a polynomial as an upper-bound for the runtime when there is no REDoS

vulnerability causing an exponential blowup. Being able to generate strings for both the cases would make the tool even more useful. The fuzzers in this regard deserve some credit, in that their brute-force approach is sometimes capable of detecting regular expressions that have unacceptable polynomial runtimes (unacceptable in the sense that the matching operation exceeds a certain runtime threshold). We believe it is quite possible to adopt our analysis (and our tool) to implement this feature, the feasibility of such an analysis is discussed in [BDvdM14], although the proposed approach there is different from ours.

As demonstrated in Chapter 6, the efficiency of the analyser compares well with that of the Microsoft SDL Regex Fuzzer [Mic11]; our tool is capable of analyzing most expressions in a matter of micro-seconds (as opposed to minutes). Given that we are computing sets of sets of states, the analysis may explore a large search space for very complex expressions. One may take some comfort from the fact that type checking and inference for functional programming languages can have high complexity in the worst case [Mai89, Sei94] that may not manifest itself in practice. Nonetheless, we aim to revisit the design of the tool and further optimize it (e.g using bit-vectors to implement sets instead of integer lists).

Pruning the search space may lead to further improvements in efficiency. An intriguing possibility is to implement the analysis on many-core graphics hardware (GPUs). Using the right data structure representation for transitions, GPUs can efficiently explore nondeterministic transitions in parallel, as demonstrated in the iNFAnt regular expression matcher [CRRS10].

Finally, the search tree logic may have independent interest and pos-

sible connections to other substructural logics such as Linear Logic [Gir87, GLT89], Separation Logic [IO01, Rey02], Lambek's syntactic calculus [Lam58], or substructural calculi for parse trees [Thi13]. Search trees are dual to parse trees in the sense that the nodes represent a disjunction rather than a conjunction.

BIBLIOGRAPHY

[Aho90] Alfred V. Aho. Algorithms for Finding Patterns in Strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (vol. A)*, pages 255–300. MIT Press, Cambridge, MA, USA, 1990. 9, 136

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA, USA, 1991. 30

[ALSU07] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques and Tools*. Addison Wesley, second edition, 2007. 4, 5, 16, 19, 21, 23, 134

[Ant96] Valentin Antimirov. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theor. Comput. Sci.*, 155(2):291–319, March 1996. 26

[AP97] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 1997. 134

[App92] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992. 30, 132

[BC08] Michela Becchi and Patrick Crowley. Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 25:1–25:12, New York, NY, USA, 2008. ACM. 28

[BDvdM14] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching. In *AFL*, pages 109–123, 2014. 131, 137

[Brz64] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, 1964. 23, 28

[BS86] G. Berry and R. Sethi. From Regular Expressions to Deterministic Automata. *Theor. Comput. Sci.*, 48(1):117–126, December 1986. 26

[CF10] Avik Chaudhuri and Jeffrey S. Foster. Symbolic Security Analysis of Ruby-on-rails Web Applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 585–594, New York, NY, USA, 2010. ACM. 32

[Cho56] Noam Chomsky. Three models for the description of language. *IRI Transactions on Information Theory*, 2(3):113–124, 1956. 2, 16

[CM04] B. Chess and G. McGraw. Static Analysis for Security. *Security & Privacy, IEEE*, 2(6):76–79, 2004. 31, 133

[Cor12] Microsoft Corporation. Microsoft security development lifecycle. Available at `http://www.microsoft.com/security/sdl/process/verification.aspx`, 2012. 12

[Cou13] Patrick Cousot. Abstract Interpretation. Available at `http://www.di.ens.fr/~cousot/AI/`, 2013. 32

[Cox07] Russ Cox. Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, Php, Python, Ruby, ...). Available at `http://swtch.com/~rsc/regexp/regexp1.html`, January 2007. 5, 7, 11, 12, 22, 23, 27, 68, 77, 130, 133

[Cox09] Russ Cox. Regular Expression Matching: The Virtual Machine Approach. Available at `http://swtch.com/~rsc/regexp/regexp2.html`, December 2009. 9, 27, 68, 77

[Cox10] Russ Cox. Regular Expression Matching in the Wild. Available at `http://swtch.com/~rsc/regexp/regexp3.html`, March 2010. 23, 27, 136

[Cox11] Russ Cox. Irregular expression matching with the .NET stack. Available at `http://research.swtch.com/irregexp`, May 2011. 10, 28, 136

[CRRS10] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. iNFAnt: NFA Pattern Matching on GPGPU Devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, October 2010. 28, 137

[CW03] Scott A. Crosby and Dan S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003. 133

[CZ02] J.-M. Champarnaud and D. Ziadi. Canonical Derivatives, Partial Derivatives and Finite Automaton Constructions. *Theor. Comput. Sci.*, 289(1):137–163, October 2002. 27

[DMS06] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison Wesley, 2006. 31

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at Work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '01, pages 162–174, New York, NY, USA, 2001. ACM. 10, 31, 132, 133

[FC04] Alain Frisch and Luca Cardelli. Greedy Regular Expression Matching. In *ICALP*, pages 618–629, 2004. 28, 132, 133

[FF86] Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts*, pages 193–217. North-Holland, 1986. 30, 132

[Fou12] Python Software Foundation. re - Regular expression operations. Available at `https://docs.python.org/2/library/re.html`, 2012. 9

[Gir87] Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987. 133, 138

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proof and Types*. Cambridge University Press, 1989. 133, 138

[Goy09a] Jan Goyvaerts. Runaway Regular Expressions: Catastrophic Backtracking. Available at `http://www.regular-expressions.info/catastrophic.html`, 2009. 11, 12

[Goy09b] Jan Goyvaerts. Runaway Regular Expressions: Catastrophic Backtracking. Available at `http://www.`

regular-expressions.info/catastrophic.html, 2009. 130

[GS07] Gregor Gramlich and Georg Schnitger. Minimizing Nfa's and Regular Expressions. *J. Comput. Syst. Sci.*, 73(6):908–923, September 2007. 22

[Har97] Robert Harper. Programming in Standard ML. Available at ftp://ftp.cs.cmu.edu/user/rwh/www/home/smlbook/book.pdf, 1997. 7, 31

[Har99] Robert Harper. Proof-Directed Debugging. *J. Funct. Program.*, 9(4):463–469, July 1999. 7, 10, 31, 132, 133

[Hay87] Christopher T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987. 30

[Haz12a] Philip Hazel. PCRE - Perl Compatible Regular Expressions. Available at http://www.pcre.org/, 2012. 45

[Haz12b] Philip Hazel. PCRE Change Log. Available at http://www.pcre.org/changelog.txt, 2012. 10, 45

[HN11] Fritz Henglein and Lasse Nielsen. Regular expression containment: coinductive axiomatization and computational interpretation. In *Principles of Programming Languages (POPL 2011)*, pages 385–398, 2011. 132

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, October 1969. 30

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979. 2, 4, 16, 19, 133

[HVO06] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL Injection Attacks and Countermeasures. In *ISSSE'06.* IEEE, 2006. 31

[IO01] Samin S. Ishtiaq and Peter O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *Principles of Programming Languages (POPL)*, pages 14–26. ACM, 2001. 30, 133, 138

[Kea91] Steven M. Kearns. Extending regular expressions with context operators and parse extraction. *Softw. Pract. Exper.*, 21(8):787–804, July 1991. 9

[Kle56] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, 1956. 16, 17

[KRT13] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *International Conference on Network and System Security (NSS 2013)*, number LNCS 7873 in LNCS, pages 135–148. Springer, 2013. 131

[Lam58] Joachim Lambek. The Mathematics of Sentence Structure. *American Mathematical Monthly*, 65(3):154–170, 1958. 138

[Lan64] Peter J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, January 1964. 29, 35, 132

[Lau01] V. Laurikari. Efficient submatch addressing for regular expressions. Master's thesis, Dept. Comp. Sci. Eng, Helsinki Univ, Finland, 2001. 9, 28, 135

[LL05] V.B. Livshits and M.S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, volume 14, pages 18–18, 2005. 31

[Mai89] H.G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 382–401. ACM, 1989. 137

[Mic11] Microsoft. SDL Regex Fuzzer. Available at `http://www.microsoft.com/en-gb/download/details.aspx?id=20095`, 2011. 12, 126, 130, 137

[MY60] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, March 1960. 17, 22

[NN10] Kedar Namjoshi and Girija Narlikar. Robust and Fast Pattern Matching for Intrusion Detection. In *Proceedings of the 29th Conference on Information Communications*, INFOCOM'10, pages 740–748, Piscataway, NJ, USA, 2010. IEEE Press. 11

[NVI11] NVIDIA. What is CUDA? Available at http://www.nvidia.com/object/what_is_cuda_new.html, 2011. 28

[ORT09] Scott Owens, John Reppy, and Aaron Turon. Regular-expression Derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009. 22, 28

[OWA12] The Open Web Application Security Project OWASP. Regular Expression Denial of Service - ReDoS. Available at https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS, 2012. 11, 12, 130

[Per05] PerlMonks.org. Benchmarks aren't everything. Available at http://perlmonks.org/index.pl?node_id=502408, October 2005. 28

[Per07] PerlMonks.org. Perl regexp matching is slow?? Available at http://www.perlmonks.org/?node_id=597262, 2007. 28

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. 29

[Reg12] RegExLib.com. Regular Expression Library. Available at http://regexlib.com/, 2012. 120

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, August 1972. 30, 31

[Rey99] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, New York, NY, USA, 1999. 29

[Rey02] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002. 30, 56, 132, 133, 138

[RT11] Asiri Rathnayake and Hayo Thielecke. Regular Expression Matching and Operational Semantics. In *Structural Operational Semantics (SOS 2011)*, volume 62 of *EPTCS*, pages 31–45, 2011. 28, 30

[RW12] Alex Roichman and Adar Weidman. Regular Expression Denial of Service. Available at http://www.checkmarx.com/white_papers/redos-regular-expression-denial-of-service/, 2012. 11, 12

[Sei94] Helmut Seidl. Haskell Overloading is DEXPTIME-Complete. *Information Processing Letters*, 52(2):57–60, 1994. 137

[SEJ06] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking Algorithmic Complexity Attacks Against a NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 89–98, Washington, DC, USA, 2006. IEEE Computer Society. 11, 133

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. 2, 4

[SK02] Axel Simon and Andy King. Analyzing String Buffers in C. In *AMAST*, pages 365–379, 2002. 31

[SL12] Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives. In *PPDP*, pages 79–90, 2012. 132

[SM14] Satoshi Sugiyama and Yasuhiko Minamide. Checking Time Linearity of Regular Expression Matching Based on Backtracking. *IPSJ Transactions on Programming (to appear)*, 2014. 131

[Sou12] Sourcefire. Snort (IDS/IPS). Available at `http://www.snort.org/`, 2012. 11, 120

[Sul12] Martin Sulzmann. Regular Expression Sub-Matching using Partial Derivatives. Available at `http://sulzmann.blogspot.co.uk/2012/06/regular-expression-sub-matching-using.html`, June 2012. 28, 136

[Sul13] Bryan Sullivan. Regular Expression Denial of Service Attacks and Defenses - MSDN. Available at `http://msdn.microsoft.com/en-us/magazine/ff646973.aspx`, 2013. 12, 130

[Thi99] Hayo Thielecke. Continuations, functions and jumps. *SIGACT News*, 30(2):33–42, June 1999. 30

[Thi13] Hayo Thielecke. On the Semantics of Parsing Actions. *Science of Computer Programming*, 2013. 138

[Tho68] Ken Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422, June 1968. 5, 19, 22, 26, 27, 77

[TR14] Hayo Thielecke and Asiri Rathnayake. RXXR2: Regular expression denial of service (REDoS) static analysis. Available at `http://www.cs.bham.ac.uk/~hxt/research/rxxr2/index.shtml`, 2014. 117, 121, 124

[Tra08] Python Bug Tracker. Issue 2537: re.compile(...) should not fail. Available at `http://bugs.python.org/issue2537`, 2008. 124

[Wat94] Bruce W. Watson. A Taxonomy of Finite Automata Cconstruction Algorithmes. Computing science note, Eindhoven University of Technology, The Netherlands, 1994. 19

[YCD+06] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and Memory-efficient Regular Expression Matching for Deep Packet Inspection. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, pages 93–102, New York, NY, USA, 2006. ACM. 21

[YMH+12] Liu Yang, Pratyusa Manadhata, William Horne, Prasad Rao, and Vinod Ganapathy. Fast submatch extraction using obdds. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 163–174, New York, NY, USA, 2012. ACM. 9