Certified Virtual Machine-Based Regular Expression Parsing

Thales Antônio Delfino

A Dissertation submitted for the degree of Master in Computer Science.

Departamento de Computação
Universidade Federal de Ouro Preto

June 20, 2019

Contents

D349c Delfino, Thales Antônio.

Certified virtual machine-based regular expression parsing [manuscrito] / Thales Antônio Delfino. - 2019.

69f.: il.: color; grafs; tabs; Código-fonte.

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro.

Dissertação (Mestrado) - Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação. Programa de Pós-Graduação em Ciência da Computação.

Área de Concentração: Ciência da Computação.

1. VM/CMS (Sistema operacional de computador). 2. Análise de redes . 3. Certificado digital. 4. Algoritmos. I. Ribeiro, Rodrigo Geraldo. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004.423.24

Catalogação: www.sisbin.ufop.br

UFOP

ATA DE DEFESA DE DISSERTAÇÃO

Aos 26 dias do mês de abril do ano de 2019, às 14:00 horas, nas dependências do Departamento de Computação (Decom), foi instalada a sessão pública para a defesa de dissertação do mestrando **Thales Antonio Delfino**, sendo a banca examinadora composta pelo Prof. Dr. Rodrigo Geraldo Ribeiro (Presidente - UFOP), pelo Prof. Dr. Jose Romildo Malaquias (Membro - UFOP), pelo Prof. Dr. Leonardo Vieira dos Santos Reis (Membro - Externo). Dando início aos trabalhos, o presidente, com base no regulamento do curso e nas normas que regem as sessões de defesa de dissertação, concedeu ao mestrando 60 minutos para apresentação do seu trabalho intitulado "Certified Virtual Machine-Based Regular Expression Parsing". Terminada a exposição, o presidente da banca examinadora concedeu, a cada membro, um tempo máximo de 60 minutos para perguntas e respostas ao candidato sobre o conteúdo da dissertação, na seguinte ordem: Primeiro, Prof. Leonardo Vieira dos Santos Reis; segundo, Prof. Jose Romildo Malaquias; terceiro, Prof. Rodrigo Geraldo Ribeiro. Dando continuidade, ainda de acordo com as normas que regem a sessão, o presidente solicitou aos presentes que se retirassem do recinto para que a banca examinadora procedesse à análise e decisão, anunciando, a seguir, publicamente, que o mestrando foi aprovado por unanimidade, sob a condição de que a versão definitiva da dissertação deva incorporar todas as exigências da banca, devendo o exemplar final ser entregue no prazo máximo de 6 (seis) meses à Coordenação do Programa. Para constar, foi lavrada a presente ata que, após aprovada, vai assinada pelos membros da banca examinadora e pelo mestrando. Ouro Preto, 26 de abril de 2019.

Rodrigo Geraldo Ribeiro

Prof. Dr. Rodrigo Geraldo Ribeiro

Presidente

José Romildo Malaquias

Prof. Dr. Jose Romildo Malaquias

Leonardo Vieira dos Santos Reis

Prof. Dr. Leonardo Vieira dos Santos Reis

[illegible]

Mestrando

Abstract

Regular expressions (REs) are pervasive in computing. We use REs in text editors, string search tools (like GNU-Grep) and lexical analyzers generators. Most of these tools rely on converting REs to its corresponding finite state machine or use REs derivatives for directly parse an input string. In this work, we investigated the suitability of another approach: instead of using derivatives or generating a finite state machine for a given RE, we developed a certified virtual machine-based algorithm (VM) for parsing REs, in such a way that a RE is merely a program executed by the VM over the input string. First, we developed a small-step operational semantics for the algorithm, implemented it in Haskell, tested it using QuickCheck and provided proof sketches of its correctness with respect to RE standard inductive semantics. After that, we developed a big-step operational semantics, an evolution of the small-step one. Unlike the small-step, the big-step semantics can deal with problematic REs. We showed that the big-step semantics for our VM is also sound and complete with regard to a standard inductive semantics for REs and that the evidence produced by it denotes a valid parsing result. All of our results regarding the big-step semantics are formalized in Coq proof assistant and, from it, we extracted a certified algorithm, which we used to build a RE parsing tool using Haskell programming language. Experiments comparing the efficiency of our algorithm with another Haskell tool are reported.

1. Introduction

Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little.

Bertrand Meyer, Designer of Eiffel Programming Language.

We name parsing the process of analyzing if a sequence of symbols matches a given set of rules. Such rules are usually specified in a formal notation, like a grammar. If a string can be obtained from those rules, we have success: we can build some evidence that the input is in the language described by the underlying formalism. Otherwise, we have a failure: no such evidence exists.

In this work, we focus on the parsing problem for regular expressions (REs), which are an algebraic and compact way of defining regular languages (RLs), i.e., languages that can be recognized by (non-)deterministic finite automata and equivalent formalisms. REs are widely used in string search tools, lexical analyzer generators and XML schema languages [21]. Since RE parsing is pervasive in computing, its correctness is crucial. Nowadays, with the recent development of languages with dependent types and proof assistants it has become possible to represent algorithmic properties as program types which are verified by the compiler. The usage of proof assistants to verify RE parsing / matching algorithms were the subject of study of several recent research works (e.g [18, 47, 35, 3]).

Approaches for RE parsing can use representations of finite state machines (e.g. [18]), derivatives (e.g. [47, 36, 35]) or the so-called pointed RE's or its variants [3, 19]. Another approach for parsing is based on parsing machines, which dates back to 70's with Knuth's work on top-down syntax analysis for context-free languages [31]. Recently, some works have tried to revive the use of such machines for parsing: Cox [12] defined a VM for which a RE can be seen as "high-level programs" that can be compiled to a sequence of such VM instructions and Lua library LPEG [27] defines a VM whose instruction set can be used to compile Parser Expressions Grammars (PEGs) [20]. Such renewed research interest is motivated by the fact that is possible to include new features by just adding and implementing new machine instructions.

Since LPEG VM is designed with PEGs in mind, it is not appropriate for RE parsing, since the "star" operator for PEGs has a greedy semantics which differs from the conventional RE semantics for this operator. Also, Cox's work on VM-based RE parsing has problems. First, it is poorly specified: both the VM semantics and the RE compilation process are described only informally and no correctness guarantees are even mentioned. Second, it does not provide an evidence for matching, which could be used to characterize a disambiguation strategy, like Greedy [21] and POSIX [50]. To the best of our knowledge, no previous work has formally defined a VM for RE parsing that produces evidence (parse trees) for successful matches. The objective of this work is to give a first step in filling this gap. More specifically, we are interested in formally specify and prove the correctness of a VM based semantics for RE parsing which produces bit-codes as a memory efficient representation of parse-trees. As pointed by [42], bit-codes are useful because they are not only smaller than the parse tree, but also smaller than the string being parsed and they can be combined with methods for text compression. We would like to emphasize that, unlike Cox's work, which develop its VM using a instruction set like syntax and semantics, we use, as inspiration, VMs for the λ-calculus, like the SECD and Krivine machines [32, 33].

One important issue regarding RE parsing is how to deal with the so-called problematic RE[1][21]. In order to avoid the well-known issues with problematic RE, we use a transformation proposed by Medeiros et. al. [41] which turns a problematic RE into an equivalent non-problematic one. We proved that this algorithm indeed produces equivalent REs using Coq proof assistant.

1.1. Objectives

The main objective of this dissertation is to develop a VM-based RE parsing algorithm and formally verify its relevant correctness properties (completeness and soundness with standard RE semantics[2].)

1.2. Contributions

Our contributions are:

- We present a small-step semantics for RE inspired by Thompson's NFA[3] construction [51]. The main novelty of this presentation is the use of data-type derivatives, a well-known concept in functional programming community, to represent the context in which the current RE being evaluated occur. We show informal proofs[4] that our semantics is sound and complete with respect to RE inductive semantics.

[1]We say that a RE e is problematic if there is e_1 s.t. $e = e_1^\star$ and e_1 accepts the empty string.

[2]We say that the VM semantics is sound with respect to standard RE semantics if, and only if, every string accepted by the VM is also accepted by the RE semantics. In the other hand, we say that a VM semantics is complete if, and only if, all strings accepted by the RE semantics are also accepted by the VM.

[3]Non-deterministic finite automata.

[4]By "informal proofs" we mean proofs that are not mechanized in a proof-assistant.

- We describe a prototype implementation of our semantics in Haskell and use QuickCheck [11] to test our semantics against a simple implementation of RE parsing, presented in [19], which we prove correct in the Appendix A.
- We show how our proposed semantics can produce bit codes that denote parse trees [42] and test that such generated codes correspond to valid parsing evidence using QuickCheck. Our test cases cover both accepted and rejected strings for randomly generated REs.
- We develop a certified implementation of an algorithm that converts a problematic RE into a non-problematic one [41].
- We present a big-step operational semantics, that uses the above mentioned algorithm to deal correctly with problematic RE (unlike our previous small-step semantics) and also produces bit-codes as parsing evidence.
- We prove that the bit-codes produced by our semantics are valid parsing evidence.
- We extract from our formalization a certified algorithm in Haskell and used it to build a RE parsing tool. We compare its performance against a well-known Haskell library for RE parsing.

1.3. Published Material

This dissertation is based on two papers: one was published in a peer-reviewed conference and the other was submitted to a journal.

- “Towards Certified Virtual Machine-based Regular Expression Parsing” is described in our SBLP 2018 paper [14].
- “Certified Virtual Machine Based Regular Expression Parsing” is described in a paper submitted to the Science of Computer Programming journal. This paper is under review.

1.4. Dissertation Structure

This work is organized as follows. Chapter 2 reviews some important concepts about formal language theory and other relevant subjects to our research. We present our proposed semantic in Chapter 3. Related works are summarized in Chapter 4 and Chapter 5 concludes.

All source code produced, including instructions on how to build it, are available at [13], including the LaTeX code of this document.

2. Background

This chapter is concerned with concepts that are fundamental to this work. We start by giving a succinct review of formal language theory in Section 2.1, as found in classic textbooks [25]. Section 2.2 approaches REs. Section 2.3 presents Thompson NFA construction for REs, in which the proposed semantic of this work are inspired. The relation between RE parsing and bit-coded parse trees is shown in Section 2.4. Next, we present an introduction to formal semantics in 2.5 based in [43], and give some examples of operational semantics. Section 2.6 brings some basic notions of Haskell programming language, including an overview of QuickCheck in Section 2.6.1, as well as the concept of data-type derivatives in Section 2.6.2, a very useful resource we adopted on our small-step operational semantics. Section 2.7 concludes this chapter by presenting a succinct introduction to Coq proof assistant.

A reader familiar with these topics can safely skip this chapter.

2.1. Formal Language Theory

The whole formal language theory is centered in the notion of an alphabet, which consists of a non-empty finite set of symbols. Following common practice, we use the meta-variable Σ to denote an arbitrary alphabet. A string over Σ is a finite sequence of symbols from Σ. We let ϵ denote the empty string and if x is a string over some alphabet, notation $|x|$ denotes the length of x. We let x^n denote the string formed by n repetitions of x. When $n = 0$, $x^0 = \epsilon$. A language over an alphabet Σ is a set of strings over Σ.

Below we present examples of such concepts.

Example 1. Consider the alphabet $\Sigma = \{0, 1\}$. The following are examples of strings over Σ: $\epsilon, 0, 1, 00, 111, 0101$. Note that ϵ is a valid string for any alphabet.

Examples of languages over $\Sigma = \{0, 1\}$ are $\{0, 11, \epsilon\}$ and $\{0^n1^n \mid n \geqslant 0\}$.

Since languages are sets of strings, we can generate new languages by applying standard set operations, like intersection, union, complement, and so on [25]. In addition to standard set operations, we can build new languages using some operations over strings. Given two languages L_1 and L_2, we define the concatenation, L_1L_2, as:

$$L_1L_2 = \{xy \mid x \in L_1 \land y \in L_2\}$$

Using concatenation, we can define the iterated concatenation as:

$$\begin{array}{lcl} L^0 & = & \{\epsilon\} \\ L^{n+1} & = & L^n L \end{array}$$

Finally, the Kleene closure operator of a language L, $L^\star$, can be defined as:

$$L^\star = \bigcup_{n \in \mathbb{N}} L^n$$

Given an alphabet Σ, $\Sigma^\star$ denote the set of all possible strings formed using symbols from Σ.

Another pervasive notion in formal language theory is the so-called Deterministic finite state automata (DFAs).

Definition 1. A deterministic finite automata (DFA) M is a 5-tuple $M = (S, \Sigma, \delta, i, F)$, where:

- S: non-empty, finite set of states.
- Σ: input alphabet.
- $\delta : S \times \Sigma \to S$: transition function.
- $i \in S$: initial state.
- $F \subseteq S$: set of final states.

In order to define the set of strings accepted by a DFA, we need to extend its transition function to operate on strings and not only on symbols of its input alphabet as follows:

$$\begin{array}{lcl} \widehat{\delta}(s, \epsilon) & = & s \\ \widehat{\delta}(s, ay) & = & \widehat{\delta}(\delta(s, a), y) \end{array}$$

with $s \in S$, $a \in \Sigma$ and $y \in \Sigma^\star$. Using this extended transition function we can define the language accepted by a DFA M as:

$$L(M) = \{w \in \Sigma^\star \mid \widehat{\delta}(i, w) \in F\}$$

Example 2. Consider the following language

$$L = \{w \in \{0,1\}^\star \mid w \text{ starts with a 0 and ends with a 1}\}$$

A DFA that accepts L is presented in Figure 2.1. From this state diagram, the state set S and the final states F are obvious. The Table 2.1 shows the transition function for that DFA.

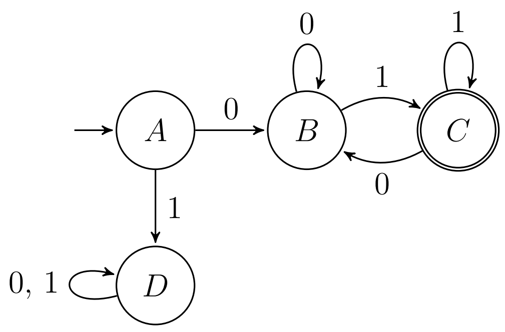


Figure 2.1.: DFA for $L = \{w \in \{0,1\}^\star \mid w \text{ starts with a 0 and ends with a 1}\}$

δ	0	1
A	B	D
B	B	C
C	B	C
D	D	D

Table 2.1.: Transition function for the previous DFA

2.2. Regular Expressions

REs are an algebraic and widely used formalism for specifying languages in computer science. In this section we will look at the formal syntax and semantics for REs.

Definition 2 (RE syntax)**.** Let Σ be an alphabet. The set of REs over Σ is described by the following grammar:

$$\begin{array}{rcl} e & \rightarrow & \emptyset \\ & | & \epsilon \\ & | & a \\ & | & e\,e \\ & | & e + e \\ & | & e^\star \end{array}$$

where ϵ represents an empty RE; $a \in \Sigma$; the meta-variable e denotes an arbitrary RE; "ee" means the concatenation of two REs; "$e{+}e$" represents the choice operator between two REs and "$e^\star$" is the Kleene closure of a RE e.

A RE describes a set of strings. This is captured by the following definition:

Definition 3 (RE functional semantics)**.** Let Σ be an alphabet. We define the semantics of a RE over Σ using the following function, $[\![\_]\!] : RE \rightarrow \mathcal{P}(\Sigma^\star)$, in which $\mathcal{P}(x)$ denotes the powerset of a set x:

$$\begin{array}{lcl}
[\![\emptyset]\!] & = & \emptyset \\
[\![\epsilon]\!] & = & \{\epsilon\} \\
[\![a]\!] & = & \{a\} \\
[\![e\,e']\!] & = & [\![e]\!]\,[\![e']\!] \\
[\![e+e']\!] & = & [\![e]\!] \cup [\![e']\!] \\
[\![e^\star]\!] & = & ([\![e]\!])^\star
\end{array}$$

After a precise characterization of RE, we can now use it to define the class of Regular Languages (RLs).

Definition 4 (Regular language)**.** A language $L \subseteq \Sigma^\star$ is a RL if there is an RE e such that $L = [\![e]\!]$.

In order to clarify the previous definitions, we present some examples of REs and describe their meaning.

Example 3. Consider $\Sigma = \{0, 1\}$.

- The RE $e = 0^\star 1 0^\star$ denotes the following language

$$L = \{w \in \{0,1\}^\star \mid w \text{ has just one occurrence of } 1\}$$

- The RE $e = (1+\epsilon)0$ denotes the language $L = \{10, 0\}$.
- The RE $e = \emptyset^\star$ denotes the language $L = \{\epsilon\}$.
- The RE $e = 0(0+1)^\star 1$ denotes the language

$$L = \{w \in \{0,1\}^\star \mid w \text{ starts with a } 0 \text{ and ends with a } 1\}$$

Meta-variable e will denote an arbitrary RE and a an arbitrary alphabet symbol. As usual, all meta-variables can appear primed or subscripted. In our Coq formalization, we represent alphabet symbols using type `ascii` and our Haskell implementation represents alphabet symbols using `Char` type. We let concatenation of RE, strings and lists by juxtaposition. Given a RE, we let its `size` be defined by the following function:

$$\begin{array}{lcl}
\texttt{size}(\emptyset) & = & 0 \\
\texttt{size}(\epsilon) & = & 1 \\
\texttt{size}(a) & = & 2 \\
\texttt{size}(e_1 + e_2) & = & 1 + \texttt{size}(e_1) + \texttt{size}(e_2) \\
\texttt{size}(e_1\,e_2) & = & 1 + \texttt{size}(e_1) + \texttt{size}(e_2) \\
\texttt{size}(e^\star) & = & 1 + \texttt{size}(e)
\end{array}$$

Given a pair (e, s), formed by a RE expression e and a string s, we define its complexity as $(\texttt{size}(e), |s|)$. Many proofs are made by well-founded induction over the complexity of that pair.

Following common practice [35, 47, 46], we adopt an inductive characterization of RE membership semantics as shown in Figure 2.2. We let judgment $s \in [\![e]\!]$ denote that string s is in the language denoted by RE e.

$$\frac{}{\epsilon \in \epsilon}\{\mathit{Eps}\} \qquad \frac{a \in \Sigma}{a \in a}\{\mathit{Chr}\}$$

$$\frac{s \in e}{s \in e + e'}\{\mathit{Left}\} \qquad \frac{s' \in e'}{s' \in e + e'}\{\mathit{Right}\}$$

$$\frac{}{\epsilon \in e^\star}\{\mathit{StarBase}\} \qquad \frac{s \in e \quad s' \in e^\star}{ss' \in e^\star}\{\mathit{StarRec}\}$$

$$\frac{s \in e \quad s' \in e'}{ss' \in ee'}\{\mathit{Cat}\}$$

Figure 2.2.: RE inductive semantics.

Rule *Eps* states that the empty string (denoted by the ϵ) is in the language of RE ϵ. For any single character a, the singleton string a is in the language of RE a. Given membership proofs for REs e and e', $s \in [\![e]\!]$ and $s' \in [\![e']\!]$, rule *Cat* can be used to build a proof for the concatenation of these REs. Rule *Left* (*Right*) creates a membership proof for $e + e'$ from a proof for e (e'). Semantics for Kleene star is built using the following well known equivalence of REs: $e^\star = \epsilon + e\,e^\star$.

Next, we present a simple example of the inductive RE semantics.

Example 4. The string aab is in the language of RE $(aa+b)^\star$, as the following derivation shows:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \in \Sigma}{a \in a}\mathit{Chr} \quad \dfrac{a \in \Sigma}{a \in a}\mathit{Chr}}{aa \in aa}\mathit{Cat}}{aa \in aa + b}\mathit{Left} \quad \dfrac{\dfrac{\dfrac{b \in \Sigma}{b \in b}\mathit{Chr}}{b \in aa + b}\mathit{Right} \quad \dfrac{}{\epsilon \in (aa + b)^\star}\mathit{StarBase}}{b \in (aa + b)^\star}\mathit{StarRec}}{aab \in (aa + b)^\star}\mathit{StarRec}$$

As one would expect, the inductive and functional semantic of REs are equivalent, as shown in the next theorem.

Theorem 1. *For all RE e and strings $s \in \Sigma^\star$, $s \in [\![e]\!]$ if, and only if, $s \in e$.*

Proof. Let e and s be an arbitrary RE and string, respectively.

$(\rightarrow)$: Suppose that $s \in [\![e]\!]$. We proceed by induction on the structure of e.

- Case $e = \emptyset$. We have:

$$\begin{aligned} s \in [\![\emptyset]\!] &\leftrightarrow \\ s \in \emptyset &\leftrightarrow \\ \bot \end{aligned}$$

  which makes the conclusion hold by contradiction.
- Case $e = \epsilon$. We have

$$\begin{aligned} s \in [\![\epsilon]\!] &\leftrightarrow \\ s \in \epsilon \end{aligned}$$

  Since $e = \epsilon$ and $s \in [\![\epsilon]\!]$, we have that $s = \epsilon$ and the conclusion holds by rule *Eps*.
- Case $e = a$, $a \in \Sigma$. We have:

$$\begin{aligned} s \in [\![a]\!] &\leftrightarrow \\ s \in a \end{aligned}$$

  Since $e = a$ and $s \in a$, we have that $s = a$ and the conclusion follows by rule *Chr*.
- Case $e = e_1\,e_2$. By the definition of the functional semantics, if $s \in [\![e_1\,e_2]\!]$, then exists $s_1, s_2 \in \Sigma^\star$, such that $s_1 \in [\![e_1]\!]$, $s_2 \in [\![e_2]\!]$ and $s = s_1\,s_2$. By the induction hypothesis, we have that $s_1 \in e_1$ and $s_2 \in e_2$ and the conclusion follows by using rule *Cat*.
- Case $e = e_1 + e_2$. By the definition of the functional semantics, if $s \in [\![e_1 + e_2]\!]$, then $s \in [\![e_1]\!]$ or $s \in [\![e_2]\!]$. Consider the cases:
  * Case $s \in [\![e_1]\!]$: The conclusion follows by the induction hypothesis and rule *Left*.
  * Case $s \in [\![e_2]\!]$: The conclusion follows by the induction hypothesis and rule *Right*.
- Case $e = (e_1)^\star$. Here we proceed by strong induction on the structure of s. Consider the following cases:
  * $s = \epsilon$: In this case the conclusion follows by rule *StarBase*.
  * $s \neq \epsilon$: Since $s \in ([\![(e_1)]\!])^\star$, by the definition of the Kleene closure, we have that there exists $s_1, s_2 \in \Sigma^\star$ such that $s_1 \in [\![e_1]\!]$, $s_2 \in ([\![e_1]\!])^\star$ and $s = s_1\,s_2$. The conclusion follows by the induction hypothesis and the rule *StarRec*.

($\leftarrow$) : Suppose that $s \in e$. We proceed by induction on the derivation of $s \in e$ by doing case analysis on the last rule employed to deduce $s \in e$.

- Case *Eps*: We have that $s = \epsilon$ and $e = \epsilon$. The conclusion follows by the definition of the functional semantics.
- Case *Chr*: We have that $s = a = e$. The conclusion follows by the definition of the functional semantics.
- Case *Cat*: Since the last rule used to deduce $s \in e$ was *Cat*, we have that must exists $s_1, s_2 \in \Sigma^\star$, e_1, e_2 such that $e = e_1\, e_2$, $s = s_1\, s_2$, $s_1 \in e_1$ and $s_2 \in e_2$. By the induction hypothesis, we have that $s_1 \in [\![e_1]\!]$ and $s_2 \in [\![e_2]\!]$. The conclusion follows by the definition of the functional semantics.
- Case *Left*: Since the last rule used to deduce $s \in e$ was *Left*, we have that must exists e_1, e_2 such that $e = e_1 + e_2$ and $s \in e_1$. The conclusion follows by the definition of functional semantics and the induction hypothesis.
- Case *Right*: Since the last rule used to deduce $s \in e$ was *Right*, we have that must exists e_1, e_2 such that $e = e_1 + e_2$ and $s \in e_2$. The conclusion follows by the definition of functional semantics and the induction hypothesis.
- Case *StarBase*: Since the last rule used to deduce $s \in e$ was *StarBase*, we have that $s = \epsilon$ and that exists e_1 such that $e = e_1^\star$. The conclusion follows by the definition of functional semantics and the Kleene closure operator.
- Case *StarRec*: Since the last rule used to deduce $s \in e$ was *StarRec*, we have that must exists $s_1, s_2 \in \Sigma^*$, e_1 such that $e = e_1^\star$, $s = s_1\, s_2$, $s_1 \in e_1$ and $s_2 \in (e_1)^\star$. By the induction hypothesis, we have that $s_1 \in [\![e_1]\!]$ and $s_2 \in [\![(e_1)^\star]\!]$ and the conclusion follows from the definition of the functional semantics.

□

Using the semantics for RE, we can define formally when two REs are equivalent as follows.

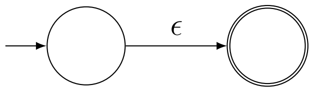
Definition 5. Let e and e' be two REs over Σ. We say e is equivalent to e', written $e \approx e'$, if the following holds:

$$\forall w. w \in \Sigma^\star \to w \in [\![e]\!] \leftrightarrow w \in [\![e']\!]$$

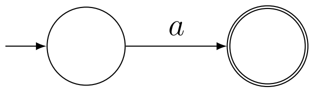
Definition 6 (Problematic REs)**.** Let e be a RE. We say that e is problematic if exists an e', such that $\epsilon \in [\![e']\!]$ and $e = e'^\star$. Otherwise, we say that e is unproblematic.

2.3. Thompson NFA Construction

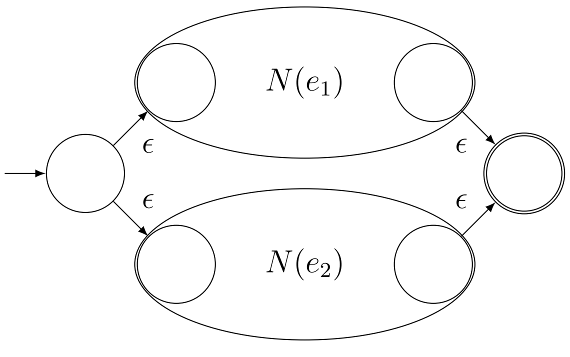
The Thompson NFA construction is a classical algorithm for building an equivalent NFA with ϵ-transitions by induction over the structure of an input RE. We follow a presentation given in [2] where $N(e)$ denotes the NFA equivalent to RE e. The construction proceeds as follows. If $e = \epsilon$, we can build the following NFA equivalent to e.



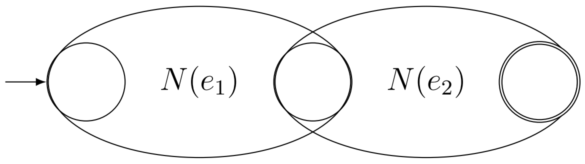
If $e = a$, for $a \in \Sigma$, we can make a NFA with a single transition consuming a:



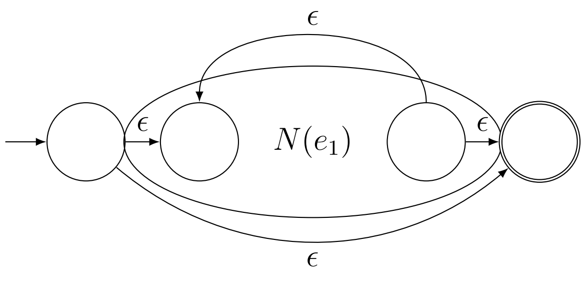
When $e = e_1 + e_2$, we let $N(e_1)$ be the NFA for e_1 and $N(e_2)$ the NFA for e_2. The NFA for $e_1 + e_2$ is built by adding a new initial and accepting state which can be combined with $N(e_1)$ and $N(e_2)$ using ϵ-transitions as shown in the next picture.



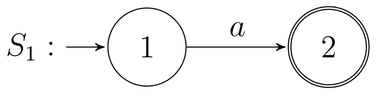
The NFA for the concatenation $e = e_1e_2$ is built from the NFAs $N(e_1)$ and $N(e_2)$. The accepting state of $N(e_1e_2)$ will be the accepting state from $N(e_2)$ and the starting state of $N(e_1)$ will be the initial state of $N(e_1)$.



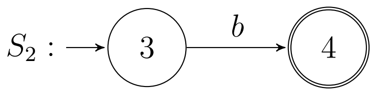
Finally, for the Kleene star operator, we built a NFA for the RE e, add a new starting and accepting states and the necessary ϵ transitions, as shown below.



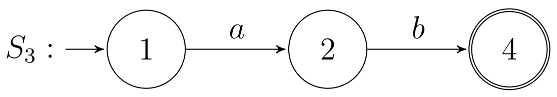
Example 5. In order to show a step-by-step automaton construction following Thompson's algorithm, we take as example the RE $((ab) + c)^*$ over the alphabet $\Sigma = \{a, b, c\}$.
The first step is to construct an automaton (S_1) that accepts the symbol a.



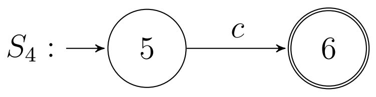
Then, we construct another automaton (S_2) that accepts the symbol b:



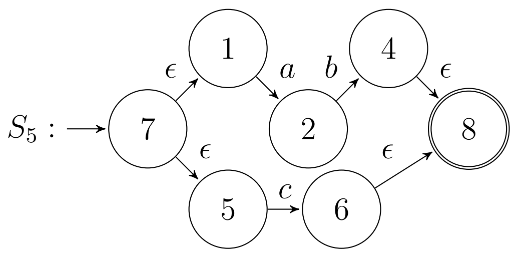
The concatenation ab is accepted by automaton S_3:



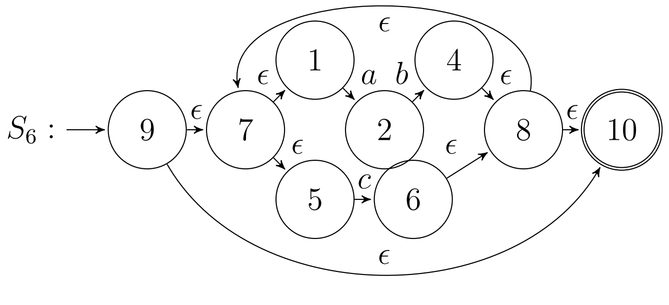
Now we build automaton S_4, which recognizes the symbol c:



The automaton S_5 accepts the RE $(ab) + c$:



Finally, we have the NFA S_6, that accepts $((ab) + c)^*$:



Originally, Thompson formulated his construction as a IBM 7094 program [51]. In our work, we reformulate it as a small-step operational semantics using contexts, modeled as data-type derivatives for RE, as shown in Section 3.1.

2.4. RE Parsing and Bit-coded Parse Trees

One way to represent parsing evidence is to build a tree that denotes a RE membership proof. Following Frisch et. al. [21] and Nielsen et. al. [42]. We let parse trees be terms whose type is its underlying RE. The following context-free grammar defines the syntax of parse trees. We use a Haskell-like syntax for lists, in which ts represents a list of elements t.

$$t \;\to\; () \mid a \mid \texttt{inl}\, t \mid \texttt{inr}\, t \mid \langle t,\, t\rangle \mid [\,] \mid t : ts$$

Figure 2.3.: Parse trees for REs.

Term () denotes the parse tree for ϵ and a the tree for a single character RE. Constructor `inl` (`inr`) tags parse trees for the left (right) operand in a union RE. A parse tree for the concatenation $e\,e'$ is a pair formed by a tree for e and another for e'. A parse tree for $e^\star$ is a list of trees for RE e. Such relationship between trees and RE is formalized by typing judgment $\vdash t : e$, which specifies that t is a parse tree for the RE e. The typing judgment is defined in Figure 2.4.

$$\frac{}{\vdash () : \epsilon} \qquad \frac{}{\vdash a : a}$$

$$\frac{\vdash t : e}{\vdash \texttt{inl}\;\, t : e + e'} \qquad \frac{\vdash t : e'}{\vdash \texttt{inr}\;\, t : e + e'}$$

$$\frac{\vdash t_1 : e_1 \quad \vdash t_2 : e_2}{\vdash \langle t_1, t_2\rangle : e_1\, e_2} \qquad \frac{}{\vdash [\,] : e^\star}$$

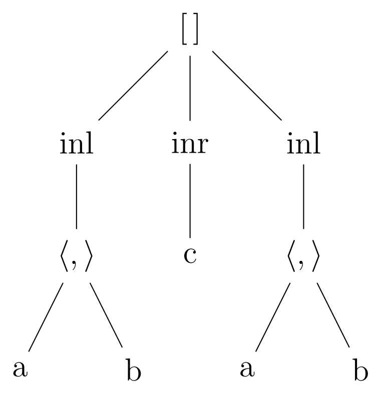
$$\frac{\vdash t : e \quad \vdash ts : e^\star}{\vdash t : ts : e^\star}$$

Figure 2.4.: Parse tree typing relation.

For any parse tree t, we can produce its parsed string using function `flatten`, which is defined below:

$$\begin{array}{lcl}
\texttt{flatten}\,() & = & \epsilon \\
\texttt{flatten}\,a & = & a \\
\texttt{flatten}\,(\texttt{inl}\,t) & = & \texttt{flatten}\,t \\
\texttt{flatten}\,(\texttt{inr}\,t) & = & \texttt{flatten}\,t \\
\texttt{flatten}\,\langle t_1, t_2\rangle & = & (\texttt{flatten}\,t_1)(\texttt{flatten}\,t_2) \\
\texttt{flatten}\,[\,] & = & \epsilon \\
\texttt{flatten}\,(t : ts) & = & (\texttt{flatten}\,t)(\texttt{flatten}\,ts)
\end{array}$$

Example 6. Consider the RE $((ab) + c)^*$ and the string *abcab*, which is accepted by that RE. Here is shown the string's corresponding parse tree:



The next theorems relates parse trees and RE semantics. The first one can be proved by an easy induction on the RE semantics derivation and the second by induction on the derivation of $\vdash t : e$.

Theorem 2. *For all s and e, if $s \in [\![e]\!]$ then exists a tree t such that* `flatten` $t = s$ *and* $\vdash t : e$.

Proof. Induction on the derivation of $s \in [\![e]\!]$. □

Theorem 3. *If* $\vdash t : e$ *then (*`flatten` *t)*$\in [\![e]\!]$.

Proof. Induction on the derivation of $\vdash t : e$. □

Nielsen et. al. [42] proposed the use of bit-marks to register which branch was chosen in a parse tree for union operator, +, and to delimit different matches done by Kleene star expression. Evidently, not all bit sequences correspond to valid parse trees. Ribeiro et. al. [47] showed an inductively defined relation between valid bit-codes and RE, accordingly to the encoding proposed by [42]. We let the judgment $bs \rhd e$ denote that the sequence of bits bs corresponds to a parse-tree for RE e.

$$\frac{}{[\,] \rhd \epsilon} \qquad \frac{}{[\,] \rhd a} \qquad \frac{bs \rhd e}{0_b\, bs \rhd e + e'}$$

$$\frac{bs \rhd e'}{1_b\, bs \rhd e + e'} \quad \frac{bs \rhd e \quad bs' \rhd e'}{bs\, bs' \rhd ee'} \qquad \frac{}{1_b \rhd e^\star}$$

$$\frac{bs \rhd e \quad bss \rhd e^\star}{0_b\, (bs\, bss) \rhd e^\star}$$

Figure 2.5.: Typing relation for bit-codes.

The empty string and single character RE are both represented by empty bit lists. Codes for RE $e\,e'$ are built by concatenating codes of e and e'. In RE union operator, $+$, the bit 0_b marks that the parse tree for $e + e'$ is built from e's and bit 1_b that it is built from e''s. For the Kleene star, we use bit 1_b to denote the parse tree for the empty string and bit 0_b to begin matchings of e in a parse tree for $e^\star$.

The relation between a bit-code and its underlying parse tree can be defined using functions `code` and `decode`, which generates a code for an input parse tree and builds a tree from a bit sequence, respectively.

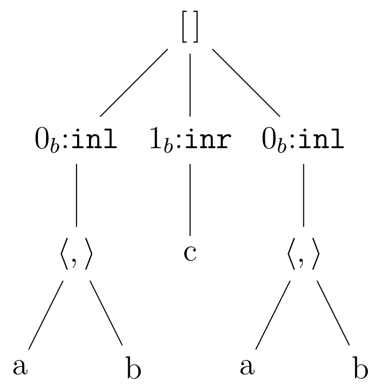
$$
\begin{array}{lcl}
\texttt{code}(() : \epsilon) & = & [\,] \\
\texttt{code}(a : a) & = & [\,] \\
\texttt{code}(\texttt{inl}\, t : e_1 + e_2) & = & 0_b\, \texttt{code}(t : e_1) \\
\texttt{code}(\texttt{inr}\, t : e_1 + e_2) & = & 1_b\, \texttt{code}(t : e_2) \\
\texttt{code}(\langle t_1, t_2 \rangle : e_1\, e_2) & = & \texttt{code}(t_1 : e_1)\, \texttt{code}(t_2 : e_2) \\
\texttt{code}([\,] : e^\star) & = & 1_b \\
\texttt{code}((t : ts) : e^\star) & = & 0_b\, \texttt{code}(t : e)\, \texttt{code}(ts : e^\star)
\end{array}
$$

Function `code` has an immediate definition by recursion on the structure of a parse tree. Note that the code generation is driven by input tree's type (i.e. its underlying RE). In the definition of function `decode`, we use an auxiliary function, `decode1`, which threads the remaining bits in recursive calls.

$$
\begin{array}{lcl}
\texttt{decode1}(bs : \epsilon) & = & ((), bs) \\
\texttt{decode1}(bs : a) & = & (a, bs) \\
\texttt{decode1}(0_b\, bs : e_1 + e_2) & = & \texttt{let}\ (t, bs_1) = \texttt{decode1}(bs : e_1) \\
 & & \texttt{in}\ (\texttt{inl}\, t, bs_1) \\
\texttt{decode1}(1_b\, bs : e_1 + e_2) & = & \texttt{let}\ (t, bs_2) = \texttt{decode1}(bs : e_2) \\
 & & \texttt{in}\ (\texttt{inr}\, t, bs_2) \\
\texttt{decode1}(bs : e_1\, e_2) & = & \texttt{let}\ (t_1, bs_1) = \texttt{decode1}(bs : e_1) \\
 & & \qquad (t_2, bs_2) = \texttt{decode1}(bs_1 : e_2) \\
 & & \texttt{in}\ (\langle t_1, t_2 \rangle, bs_2) \\
\texttt{decode1}(1_b\, bs : e^\star) & = & ([\,], bs) \\
\texttt{decode1}(0_b\, bs : e^\star) & = & \texttt{let}\ (t, bs_1) = \texttt{decode1}(bs : e) \\
 & & \qquad (ts, bs_2) = \texttt{decode1}(bs_1 : e^\star) \\
 & & \texttt{in}\ ((t : ts), bs_2) \\
 & & \\
\texttt{decode}(bs : e) & = & \texttt{let}\ (t, bs_1) = \texttt{decode1}(bs : e) \\
 & & \texttt{in}\ \texttt{if}\ bs_1 = [\,]\ \texttt{then}\ t\ \texttt{else}\ \texttt{error}
\end{array}
$$

For single character and empty string REs, its decoding consists in just building the tree and leaving the input bit-coded untouched. We build a left tree (using `inl`) for $e+e'$ if the code starts with bit 0_b. A parse tree using constructor `inr` is built whenever we find bit 1_b for a union RE. Building a tree for concatenation is done by sequencing the processing of codes for left component of concatenation and starting the processing of right component with the remaining bits from the processing of the left RE. Parsing the code for a Kleene star $e^\star$ consists in consuming a 0_b, which marks the beginning of the code for a match for e, followed for the code for a tree for e itself. We finish a list of matchings using a bit 1_b.

Example 7. We present again the same RE and string we showed in Example 6, denoted by $((ab)+c)^*$ and $abcab$, respectively. Note that the parse tree is also the same. However, this time it contains its bit codes, which are $0_b0_b0_b1_b0_b0_b1_b$. The first, third and fifth zeros in this sequence are separators and do not appear on the tree, as well as the last one digit, which defines the end of the bit codes. Remaining three digits (two zeros and one one) appear in each `inl` or `inr` on the tree.



The relation between codes and its correspondent parse trees is specified in the next theorem.

Theorem 4. *Let t be a parse tree such that $\vdash t : e$, for some RE e. Then $(\texttt{code}\, t\, e) \rhd e$ and $\texttt{decode}\,(\texttt{code}\, t\, e)\ : e = t$.*

Proof. Induction on the derivation of $\vdash t : e$. □

2.5. Formal Semantics

After defining the syntax of some formal system (e.g. a programming language), the next step in its specification is to describe its semantics [43]. There are three basic approaches to formalize semantics:

1. *Operational semantics* specifies the behavior of a programming language by defining a simple *abstract machine* for it. This machine is "abstract" in the sense that it uses the terms of the language as its machine code, rather than some low-level microprocessor instruction set. For simple languages, a state of the machine is just a term, and the machine's behavior is defined by a *transition function* that, for each state, either gives the next state by performing a step of simplification on the term or declares that the machine has halted. The *meaning* of a term t can be taken to be the final state that the machine reaches when started with `t` as its initial state. Intuitively, the operational semantics for a formal system can be seen as the mathematical specification of its interpreter.

2. *Denotational semantics* takes a more abstract view of meaning: instead of just a sequence of machine states, the meaning of a term is taken to be some mathematical object, such as a number or a function. Giving denotational semantics for a language consists of finding a collection of *semantic domains* and then defining an *interpretation function* mapping terms into elements of these domains, i.e., the denotational semantics for a programming language is a mathematical specification of its compiler.

3. *Axiomatic semantics* instead of specify how the program should behave when executed, the axiomatic semantics tries to answer the following question: "what can we prove about this program?". The axiomatic approach is concerned with logics for proving properties about some formalism which is already specified using another approach, like operational or denotational semantics.

Since our main interest is defining VM for RE parsing, we will focus on operational semantics, which is a convenient tool for specifying abstract machines of any sort. We finish this section with an example of semantics for a small language which consists solely of addition and natural numbers. While such language is certainly a toy example, it is sufficient to illustrate the main concepts used in operational semantics specifications.

2.5.1. Operational Semantics for a Simple Language

The language we will use is commonly referred in the literature as Hutton's razor [26] (HR) and serves as a minimal example to illustrate ideas in formal semantics and compilation. The HR abstract syntax is defined as follows.

Definition 7. Let n be a arbitrary numeric literal and v a variable. The abstract syntax of terms of HR is defined by the following context-free language.

$$\begin{array}{rcl} e & \to & n \\ & \mid & v \\ & \mid & e + e \end{array}$$

Following common practice, meta-variables like n, e and v can appear primed or subscripted. Next, we present some examples of terms of the HR language.

Example 8. The following are valid terms of the HR language.

- 42, denotes an integer constant.
- v_1, is a variable.
- $(v_1 + v_2) + 42$, denotes a term that sums two variables and an integer constant.

Since terms of the HR language have variables, we need to define how these should be evaluated. A possible approach is to evaluate expressions with respect to an *environment*, which will be a total function between variable names and integer values. We let the meta-variable σ denote an arbitrary environment and notation $\sigma(v)$ denotes the integer n such that $(v, n) \in \sigma$. Sometimes we write an environment as a finite mapping between variables and its corresponding integer values like $[v_1 \mapsto 1, v_2 \mapsto 2]$. In such situation, variables not explicitly listed are mapped to 0.

In operational semantic, we can use two styles to present the meaning of a formal system: the small-step and big-step style. The next sections we present semantic for HR using these styles.

Small-step Semantics for HR

Informally, a small-step operational semantics defines a method to evaluate an expression one-step at time. When considering the HR language, its small step operational semantics will be defined as a binary relation between pairs of expressions and an environment as shown in the next definition.

Definition 8. The small-step semantics for HR is the binary relation between pairs of environments and expressions defined by the rules below. We let the notation $\langle\sigma, e\rangle \rightarrow \langle\sigma, e'\rangle$ denote the pair $(\langle\sigma, e\rangle, \langle\sigma, e'\rangle)$ and symbol $\oplus$ denotes integer constant addition.

$$\boxed{\langle\sigma, e\rangle \rightarrow \langle\sigma, e'\rangle}$$

$$\frac{}{\langle\sigma, v\rangle \rightarrow \langle\sigma, \sigma(v)\rangle}\ \{VAR\} \qquad \frac{\langle\sigma, e_1\rangle \rightarrow \langle\sigma, e_1'\rangle}{\langle\sigma, e_1 + e_2\rangle \rightarrow \langle\sigma, e_1' + e_2\rangle}\ \{ADD1\}$$

$$\frac{\langle\sigma, e_2\rangle \rightarrow \langle\sigma, e_2'\rangle}{\langle\sigma, n + e_2\rangle \rightarrow \langle\sigma, n + e_2'\rangle}\ \{ADD2\} \qquad \frac{n_3 = n_1 \oplus n_2}{\langle\sigma, n_1 + n_2\rangle \rightarrow \langle\sigma, n_3\rangle}\ \{ADD3\}$$

The meaning of the previous rules are immediate. Rule $_{\{VAR\}}$ specifies that a variable evaluates to its value in the environment σ. On the other hand, rule $_{\{ADD1\}}$ specifies that the sum of two expressions e_1 and e_2 evaluates to $e_1' + e_2$, where e_1 steps to e_1' and rule $_{\{ADD2\}}$ starts the evaluation of an expression e_2 only when the first operand of a sum is completely evaluated. Finally, rule $_{\{ADD3\}}$ specifies that an expression formed by the addition of two integer constants should evaluate to their sum.

The result of evaluating a program using an operational semantics is usually called a value. In the HR language, values are just integer constants. Since a small-step semantics produces only a single pass in the program execution, we need to apply it repeatedly until we reach a value. Following standard practice, we denote the repeated application of the small-step semantics by its reflexive transitive closure, often named multi-step semantics, which is formally defined next.

Definition 9. The multi-step semantics for HR is the binary relation between pairs of environments and expressions defined as the reflexive-transitive closure of HR small-step semantics, as follows:

$$\boxed{\langle \sigma, e\rangle \rightarrow^{\star} \langle \sigma, e'\rangle}$$

$$\frac{}{\langle \sigma, e\rangle \rightarrow^{\star} \langle \sigma, e\rangle}\{Refl\} \qquad \frac{\langle \sigma, e\rangle \rightarrow \langle \sigma, e_1\rangle \quad \langle \sigma, e_1\rangle \rightarrow^{\star} \langle \sigma, e'\rangle}{\langle \sigma, e\rangle \rightarrow^{\star} \langle \sigma, e'\rangle}\{Step\}$$

Again, the meaning of the multi-step semantics is immediate. Rule *Refl* states that the relation is reflexive and rule *Step* ensures its transitivity. Next, we present an example of that semantics.

Example 9. Let $\sigma = [v_1 \mapsto 3, v_2 \mapsto 5]$ and $e = (v_1 + v_2) + 42$. Below we present part of the evaluation of e using σ.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\langle \sigma, v_1\rangle \rightarrow \langle \sigma, 3\rangle}\{VAR\}}{\langle \sigma, v_1\rangle \rightarrow \langle \sigma, 3\rangle}\{ADD1\}}{\langle \sigma, v_1 + v_2\rangle \rightarrow \langle \sigma, 3 + v_2\rangle}\{ADD1\}}{\langle \sigma, (v_1 + v_2) + 42\rangle \rightarrow \langle \sigma, (3 + v_2) + 42\rangle}\{ADD1\} \quad \cfrac{\vdots}{\langle \sigma, (3 + v_2) + 42\rangle \rightarrow^{\star} \langle \sigma, 50\rangle}\{Step\}}{\langle \sigma, (v_1 + v_2) + 42\rangle \rightarrow^{\star} \langle \sigma, 50\rangle}\{Step\}$$

The semantics of HR has some important properties: convergence, i.e. every expression can be evaluated until it reaches a value and determinism, i.e. the small-step semantics for HR is a function. Below we state theorems about the semantics and provide its proof sketches.

Theorem 5 (Determinism of HR small-step semantics)**.** *For every σ and expressions e, e' and e''; if $\langle \sigma, e\rangle \rightarrow \langle \sigma, e'\rangle$ and $\langle \sigma, e\rangle \rightarrow \langle \sigma, e''\rangle$ then e' = e".*

Proof. By induction on the structure of the derivation of $\langle \sigma, e\rangle \rightarrow \langle \sigma, e'\rangle$ and case analysis on the last rule used to conclude that $\langle \sigma, e\rangle \rightarrow \langle \sigma, e''\rangle$. $\square$

Big-step Semantics for HR

Intuitively, a big-step semantics defines a method to evaluate an expression until it reaches its final value. The big-step semantics for HR consists of a binary relation between triples formed by an environment, an expression and an integer constant. We let the notation $\langle \sigma, e\rangle \Downarrow n$ denotes the triple (σ, e, n). The next definition specifies the rules for HR big-step semantics.

$$\boxed{\langle \sigma, e\rangle \Downarrow n}$$

$$\frac{}{\langle \sigma, n\rangle \Downarrow n}\{NUM\} \qquad \frac{}{\langle \sigma, v\rangle \Downarrow \sigma(v)}\{VAR\}$$

$$\frac{\langle \sigma, e\rangle \Downarrow n \quad \langle \sigma, e'\rangle \Downarrow n'}{\langle \sigma, e + e'\rangle \Downarrow n \oplus n'}\, ADD$$

Rules NUM and VAR specifies how to evaluate numbers and variables, respectively and rule ADD say that the result of a sum expression is the addition of its corresponding numeric values. We illustrate the semantics using the following example.

Example 10. Let $\sigma = [v_1 \mapsto 3, v_2 \mapsto 5]$ and $e = (v_1 + v_2) + 42$. The evaluation of e using σ by the big-step semantics is as follows:

$$\frac{\dfrac{\dfrac{}{\langle \sigma, v_1\rangle \Downarrow 3}\{VAR\} \quad \dfrac{}{\langle \sigma, v_2\rangle \Downarrow 5}\{VAR\}}{\langle \sigma, v_1 + v_2\rangle \Downarrow 8}\{ADD\} \quad \dfrac{}{\langle \sigma, 42\rangle \Downarrow 42}\{NUM\}}{\langle \sigma, (v_1 + v_2) + 42\rangle \Downarrow 50}\{ADD\}$$

But a final question needs to be answered: How the big-step semantics relates with the small-step? The answer is given by the following theorem.

Theorem 6. *For every σ, e and n, we have that $\langle \sigma, e\rangle \Downarrow n$ if, and only if, $\langle \sigma, e\rangle \rightarrow^\star \langle \sigma, n\rangle$.*

Proof.
$(\rightarrow)$: By induction on the derivation of $\langle \sigma, e\rangle \Downarrow n$.
$(\leftarrow)$: By induction on the derivation of $\langle \sigma, e\rangle \rightarrow^\star \langle \sigma, n\rangle$. □

2.5.2. Operational Semantics for REs

Rathnayake and Thielecke [46] used operational semantics to formalize a VM-based interpreter for REs. The big-step semantics for their machine is the same as shown in Figure 2.2, differing only in notation details: instead of the symbols $\in$ and s (for instance, $s \in e$), their big-step semantics uses $\downarrow$ and w (e.g. $e \downarrow w$).

The matching of a string w to a RE e is represented by $e \downarrow w$, regarding it as a big-step operation semantics for a language with non-deterministic branching $e_1 \mid e_2$ and a non-deterministic loop e^*.

The big-step operational semantics for RE matching in the previous definition has no details about how one should attempt to match a given input string w. So, the authors defined a small-step semantics, called the *EKW machine*, that makes the matching process more explicit. The machine is named after its components: E for expression, K for continuation and W for word to be matched.

Definition 10. A configuration of the *EKWmachine* is of the form $\langle e \,;\, k \,;\, w\rangle$ where e is a RE, k is a list of REs, and w is a string. The transitions of the EKW machine are given in the next example. The accepting configuration is $\langle \epsilon \,;\, [] \,;\, \varepsilon\rangle$.

Here, e is the RE 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. 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.

Theorem 7 (Partial correctness). *$e \downarrow w$ if and only if there is a run*

$$\langle e \,;\, [] \,;\, w\rangle \to \ldots \to \langle \epsilon \,;\, [] \,;\, \varepsilon\rangle$$

Definition 11. The EKW machine transition steps are

$$\begin{array}{rcl}
\langle e_1 \mid e_2 \,;\, k \,;\, w\rangle & \to & \langle e_1 \,;\, k \,;\, w\rangle \\
\langle e_1 \mid e_2 \,;\, k \,;\, w\rangle & \to & \langle e_2 \,;\, k \,;\, w\rangle \\
\langle e_1 e_2 \,;\, k \,;\, w\rangle & \to & \langle e_1 \,;\, e_2 :: k \,;\, w\rangle \\
\langle e^* \,;\, k \,;\, w\rangle & \to & \langle e \,;\, e^* :: k \,;\, w\rangle \\
\langle e^* \,;\, k \,;\, w\rangle & \to & \langle \epsilon \,;\, k \,;\, w\rangle \\
\langle a \,;\, k \,;\, aw\rangle & \to & \langle \epsilon \,;\, k \,;\, w\rangle \\
\langle \epsilon \,;\, e :: k \,;\, w\rangle & \to & \langle e \,;\, k \,;\, w\rangle
\end{array}$$

The authors do not mention if their proposed semantics follows any disambiguation policy.

While the previous theorem ensures that all matching strings are correctly accepted, there is no guarantee that the machine accepts all strings that it should on every run. The next example will present this situation.

Example 11. Consider the RE a^{**} and the string a. A possible looping execution for the EKW machine is presented below.

$$\begin{array}{rcl}
\langle a^{**} \,;\, [] \,;\, a\rangle & \to & \langle a^* \,;\, [a^{**}] \,;\, a\rangle \\
 & \to & \langle \epsilon \,;\, [a^{**}] \,;\, a\rangle \\
 & \to & \langle a^{**} \,;\, [] \,;\, a\rangle \\
 & \to & \ldots
\end{array}$$

To solve this problem, the authors propose the PWπ machine, refining the EKW machine by the RE 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, avoiding infinite loop.

2.6. An Overview of Haskell

This section presents a brief introduction to Haskell programming language.

> Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on non-strict functional languages. (Definition of Haskell language [29])

For an introduction to the language, consider the source fragment shown in Figure 2.6. We split this section into paragraphs, in which each paragraph approaches a feature of the language.

Modules Haskell programs are composed by a sequence of *modules*. Modules provide to the programmer a way to reuse code and control namespace in programs. Each module is composed by a set of *declarations*, that may be: classes declarations, instances, datatypes and value declarations, including functions. Figure 2.6 shows a code fragment of a module called `Table` that implements operations over a table, which is represented by a list of key-value pairs. That module defines the non-functional constant `empty` and the functions `insert`, `member`, `remove` and `update` for table manipulation.

Type annotations On `Table` module, each definition is preceded by a corresponding *type annotation*.

All the names defined on `Table` module are *polymorphic*. Por instance, the `empty` constant is of type `Table a`, which is synonyms for type `[(String,a)]`. This means that `empty` can be used in contexts that require values of types that are instances of type $\forall$`a. [(String,a)]`, as for instance `[(String,Bool)]`, `[(String,Int)]`, $\forall$`a. [(String,[a])]` etc.

Functional types specify the types of the parameter and the function result (which can also be functional types). The symbol `search` has the following type annotation: `String` $\rightarrow$ `Table a` $\rightarrow$ `a`, which specifies that this function receives a value of type `Sring` as a parameter and returns a function, which receives a list of pairs made by a value of type `String` and an element of any type and returns as result an element of that type. Generally, we informally say that `search` receives two parameters (one at a "time"): a value of type `String` and a list of pairs.

It is worth noting that type annotations are, usually, optional in Haskell programs, once the compiler is able to infer the type for each expression. The process of determining the type of expressions is called *type inference*. If the programmer provides a type notation for an expression, the compiler checks if the specified definition can be of the annotated type. This verification process is called *type verification*.

```haskell
type Table a = [(String, a)]

empty :: Table a
empty = []

insert :: String -> a -> Table a -> Table a
insert s a t
| member s t = t
| otherwise = (s, a) : t

member :: String -> Table a -> Bool
member s t = not $ null [p | p <- t, fst p == s]

search :: String -> Table a -> a
search s t = snd (head [p | p <- t, fst p == s])

update :: String -> a -> Table a -> Table a
update s a [] = error "Item not found!"
update s a (x:xs)
| s == (fst x) = (s, a) : xs
| otherwise = x : update s a xs

remove :: String -> Table a -> (a, Table a)
remove s [] = error "Item not found!"
remove s (x:xs)
| s == (fst x) = (snd x, xs)
| otherwise = (fst (remove s xs), x : snd (remove s xs))
```

Figure 2.6.: A Haskell Module

Lists syntax Lists are data structures commonly used to model many problems. There is a special syntax in Haskell for representing that kind of data. The datatype `[a]` can be inductively defined as the disjoint union of an empty list, represented by `[]`, with the set of values `x:xs`, having a first element `x` of type `a` followed by a list `xs`. The symbols `[]` and `:` are *values constructors* of type list, whose types are respectively `[a]` and `a` $\rightarrow$ `[a]` $\rightarrow$ `[a]`. The use of `[a]` (instead of `List a`) is a primary form of special syntax for (types of) lists. The use of constructors `[]` and `(:)`, in which the second one is used in a infixed form, is another special notation for list construction.

Another form of special syntax for lists is shown below:

```
[True, False]
```

is an abbreviation for

```
True : (False : []).
```

On `Table` module, the function `member` uses another special syntax for lists, which is based in a commonly used notation for set definitions. That function could be defined by using sets notation as:

$$\text{member s t} = (\{\ p \mid p \in \text{t} \land (\text{fst p}) = \text{s}\} \neq \emptyset)$$

The last type of *syntactic sugar* available for lists in Haskell is succinctly presented next:

- `['a'..'z']`: list of all lowercase alphabet letters.
- `[0, 2..]`: list of natural even numbers.
- `[0..]`: list of all natural numbers.

Pattern matching A pattern is a syntactic construction that can involve the use of constants and variables introduced to define the pattern matching mechanism, which is an operation used in parameter passing. Basically, it consists simply of the fact that a constant just matches itself and a variable matches any expression. The match of a variable yields an association of the variable to the matched expression .

Pattern matching plays an important hole in the definition of functions in modern functional languages. The function `remove`, defined in `Table` module, is an example of definition that uses pattern matching over lists. This function definition is made of two alternative equations, each one specifying the correspondent result to the pattern of the received list as argument: the first equation uses the pattern `[]` and the second one uses pattern `(x:xs)`. Pattern `x:xs` is an example of a functional constant `(:)` applied to the variable `x` and the variable `xs`.

Guards The definition of function `insert` is an example of *definitions with guards*, which allows the definition of alternatives for a same equation. The alternative to be executed is the first, in the textual order, for which the guard evaluation (boolean expression) specified in the definition results in a true value.

Algebraic datatypes Figures 2.7 and 2.8 show declarations of an algebraic datatype and a function that receives values of that type as argument. The objective is to show basic features of the definition and the use of algebraic datatypes values in Haskell.

```
data Maybe a = Nothing | Just a

mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe f (Just x) = Just (f x)
mapMaybe f Nothing = Nothing
```

Figure 2.7.: Definition of an algebraic datatype and a function that uses it.

The first line illustrates the definition of an algebraic type: the keyword `data` is used on the declaration of `Maybe`. The declaration introduces `Maybe` as a *type constructor* that has two *data constructors*: `Nothing` and `Just`. The type `Maybe a` is polymorphic, i.e., universally quantified over one or more type variables. For each instanced type `t`, i.e., replaced by the type variable `a` in `Maybe a`, there is a new datatype, `Maybe t`. Values of a type `Maybe t` can be of two forms: `Nothing` or `(Just x)`, in which `x` corresponds to a value of type `t`. Data constructors can be used in patterns for decomposing values of type `Maybe t` or in expressions to build values of that type. Both cases are shown in the definition of `mapMaybe`.

Algebraic datatypes in Haskell compose a *sum of products*. The datatype definition `Tree a` indicates that a value of that type can be a leaf (`Leaf`), whose type corresponds to a trivial product of only one type, or a node built with the `Node` constructor, whose type corresponds to a product of a type `a` with two types `Tree a` (that correspond to left and right sub-trees).

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

Figure 2.8.: Algebraic datatype.

2.6.1. An Overview of QuickCheck

QuickCheck [11] is a library that allows the testing of properties expressed as Haskell functions. Such verification is done by generating random values of the desired type, instantiating the relevant property with them, and checking it directly by evaluating it to a boolean. This process continues until a counterexample is found or a specified number of cases are tested with success. The library provides generators for several standard library data types and combinators to build new generators for user-defined types.

As an example of a custom generator, consider the task of generating a random alpha-numeric character. To implement such generator, genChar, we use QuickCheck function suchThat which generates a random value which satisfies a predicate passed as argument (in example, we use isAlphaNum, which is true whenever we pass an alpha-numeric character to it), using an random generator taken as input.

```
genChar :: Gen Char
genChar = suchThat (arbitrary :: Gen Char) isAlphaNum
```

In its simplest form, a property is a boolean function. As an example, the following function states that reversing a list twice produces the same input list.

```
reverseInv : [Int] → Bool
reverseInv xs = reverse (reverse xs) ≡ xs
```

We can understand this property as been implicitly quantified universally over the argument xs. Using the function quickCheck we can test this property over randomly generated lists:

```
quickCheck reverseInv
+++ OK, passed 100 tests.
```

Test execution is aborted whenever a counter example is found for the tested property. For example, consider the following wrong property about the list reverse algorithm:

```
wrong :: [Int] → Bool
wrong xs = reverse (reverse xs) ≡ reverse xs
```

When we execute such property, a counter-example is found and printed as a result of the test.

```
quickCheck wrong
*** Failed! Falsifiable (6 tests and 4 shrinks).
[0,1]
```

2.6.2. Data-type Derivatives

The usage of evaluation contexts is standard in reduction semantics [17]. Contexts for evaluating a RE during the parse of a string s can be defined by the following context-free syntax:

$$E[\,] \to E[\,] + e \mid e + E[\,] \mid E[\,]\,e \mid e\,E[\,] \mid \star$$

The semantics of a $E[\,]$ context is a RE with a hole that needs to be "filled" to form a RE. We have two cases for union and concatenation denoting that the hole could be the left or the right component of such operators. Since the Kleene star has only a recursive occurrence, it is denoted just as a "mark" in context syntax.

Having defined our small-step semantics (Figure 3.1), we have noticed that our RE context syntax is exactly the data type for *one-hole contexts*, known as derivative of an algebraic data type. Derivatives where introduced by McBride and his coworkers [38] as a generalization of Huet's zippers for a large class of algebraic data types [1]. RE contexts are implemented by the following Haskell data-type:

```
data Hole = InChoiceL Regex | InChoiceR Regex
   | InCatL Regex | InCatR Regex | InStar
```

Constructor InChoiceL stores the right component of a union RE (similarly for InChoiceR). We need to store contexts for union because such information is used to allow backtracking in case of failure. Constructors InCatL and InCatR store the right (left) component of a concatenation and they are used to store the next subexpressions that need to be evaluated during input string parsing. Finally, InStar marks that we are currently processing an expression with a Kleene star operator.

2.7. A Tour of Coq Proof Assistant

Coq is a proof assistant based on the calculus of inductive constructions (CIC) [7], a higher-order typed λ-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called "BHK-correspondence"[1], in which types represent logical formulas, λ-terms represent proofs, and the task of checking if a piece of text is a proof of a given formula corresponds to type-checking (i.e. checking if the term that represents the proof has the type corresponding to the given formula) [48].

Writing a proof term whose type is that of a logical formula can be however a hard task, even for simple propositions. In order to make this task easier, Coq provides *tactics*, which are commands that can be used to help the user on constructing proof terms.

In this section, we provide a brief overview of Coq. We start with a small example, that uses basic features of Coq — types, functions and proof definitions. In this example, we use an inductive type that represents natural numbers in Peano notation. The `nat` type definition includes an annotation, indicating that it belongs to the `Set` sort[2]. Type `nat` is formed by two data constructors: `O`, that represents the number 0, and `S`, the successor function.

```
Inductive nat : Set :=
| O : nat
```

[1]Abbreviation of Brower, Heyting, Kolmogorov, de Bruijn and Martin-Löf Correspondence. This is also known as the Curry-Howard "isomorphism".

[2]Coq's type language classifies new inductive (and co-inductive) definitions by using sorts. `Set` is the sort of computational values (programs) and `Prop` is the sort of logical formulas and proofs.

```
| S : nat -> nat.

Fixpoint plus (n m : nat) : nat :=
   match n with
   | O => m
   | S n' => S (plus n' m)
   end.

Theorem plus0r : forall n, plus n 0 = n.
Proof.
   intros n. induction n.
   reflexivity.
   simpl. rewrite -> IHn. reflexivity.
Qed.
```

Command `Fixpoint` allows to define functions by structural recursion. The definition of `plus`, for summing two values of type `nat`, is straightforward. It should be noted that all functions defined in Coq must be total.

Besides declaring inductive types and functions, Coq allows us to define and prove theorems. In our example, we show a simple theorem about `plus`, which states that `plus n 0 = n`, for an arbitrary value `n` of type `nat`. Command `Theorem` allows the statement of a formula that we want to prove and starts the *interactive proof mode*, in which tactics can be used to produce the proof term that is the proof of such formula. In the example, various tactics are used to prove the desired result. The first tactic, `intros`, is used to move premises and universally quantified variables from the goal to the hypothesis. Tactic `induction` is used to start an inductive proof over an inductively defined object (in our example, the natural number `n`), generating a case for each constructor and an induction hypothesis for each recursive branch in constructors. Tactic `reflexivity` proves trivial equalities up to conversion and `rewrite` is used to replace terms using some equality.

For each inductively defined data type, Coq generates automatically an induction principle [7, Chapter 14]. For natural numbers, the following Coq term, called `nat_ind`, is created:

```
nat_ind
     : forall P : nat -> Prop,
      P O -> (forall n : nat, P n -> P (S n)) ->
      forall n : nat, P n
```

It expects a property (`P`) over natural numbers (a value of type `nat -> Prop`), a proof that `P` holds for zero (a value of type `P 0`) and a proof that if `P` holds for an arbitrary natural `n`, then it holds for `S n` (i.e. a value of type `forall n:nat, P n -> P (S n)`). Besides `nat_ind`, generated by the use of tactic `induction`, the term below uses the constructor of the equality type `eq_refl`, created by tactic `reflexivity`, and term `eq_ind_r`, inserted by the use of tactic `rewrite`. Term `eq_ind_r` allows concluding `P y` based on the assumptions that `P x` and `x = y` are provable.

```
Definition plus_0_r_term :=
        fun n : nat =>
                nat_ind
                   (fun n0 : nat => plus n0 O = n0) (eq_refl O)
                   (fun (n' : nat) (IHn' : plus n' O = n') =>
                      eq_ind_r (fun n0 : nat => S n0 = S n')
                               (eq_refl (S n')) IHn') n
                : forall n : nat, plus n O = n
```

Instead of using tactics, one could instead write CIC terms directly to prove theorems. This can be however a complex task, even for simple theorems like `plus_0_r`, because it generally requires detailed knowledge of the CIC type system.

An interesting feature of Coq is the possibility of defining inductive types that mix computational and logical parts. Such types are usually called *strong specifications*, since they allow the definition of functions that compute values together with a proof that this value has some desired property. As an example, consider type `sig` below, also called "subset type", that is defined in Coq's standard library as:

```
Inductive sig (A : Set)(P : A -> Prop) : Set :=
 | exist : forall x : A, P x -> sig A P.
```

Type `sig` is usually expressed in Coq by using the following syntax: $\{x : A \,|\, P\ x\}$. Constructor `exist` has two parameters. Parameter `x : A` represents the computational part. The other parameter, of type `P x`, denotes the "certificate" that `x` has the property specified by predicate `P`. As an example, consider:

```
forall n : nat, n <> 0 -> {m | n = S m}
```

This type can be used to specify a function that returns the predecessor of a natural number `n`, together with a proof that the returned value really is the predecessor of `n`. The definition of a function of type `sig` requires the specification of a logical certificate. As occurs in the case of theorems, tactics can be used when defining such functions. For example, a definition of a function that returns the predecessor of a given natural number, if it is different from zero, can be given as follows:

```
Definition predcert : forall n : nat, n <> 0 -> {m | n = S m}.
   intros n H.
   destruct n.
   destruct H. reflexivity.
   exists n. reflexivity.
Defined.
```

Tactic `destruct` is used to start a proof by case analysis on structure of a value.

Another example of a type that can be used to provide strong specifications in Coq is `sumor`, that is defined in the standard library as follows:

```
Inductive sumor(A : Set) (B : Prop) : Set :=
| inleft : A -> sumor A B
| inright : B -> sumor A B.
```

Coq standard library also provides syntactic sugar (or, in Coq's terminology, notations) for using this type: "`sumor A B`" can be written as `A + {B}`. This type can be used as the type of a function that returns either a value of type `A` or a proof that some property specified by `B` holds. As an example, we can specify the type of a function that returns a predecessor of a natural number or a proof that the given number is equal to zero as follows, using type `sumor`:

```
{p | n = S p} + {n = 0}
```

A common problem when using rich specifications for functions is the need of writing proof terms inside its definition body. A possible solution for this is to use the `refine` tactic, which allows one to specify a term with missing parts (known as "holes") to be filled latter using tactics.

The next code piece uses the `refine` tactic to build the computational part of a certified predecessor function. We use holes to mark positions where proofs are expected. Such proof obligations are later filled by tactic `reflexivity` which finishes `predcert` definition.

```
Definition predcert : forall n : nat, {p | n = S p} + {n = 0}.
  refine (fun n =>
            match n with
            | O => inright _
            | S n' => inleft _ (exist _ n' _)
            end) ; reflexivity.
Defined.
```

The same function can be defined in a more succinct way using notations introduced in [10].

```
Definition predcert : forall n : nat, {p | n = S p} + {n = 0}.
  refine (fun n =>
           match n with
           | O => !!
           | S n' => [|| n' ||]
           end) ; reflexivity.
Defined.
```

The utility of notations is to hide the writing of constructors and holes in function definitions.

Another useful type for specifications is `maybe`, which allows a proof obligation-free failure for some predicate [10].

```
Inductive maybe (A : Set) (P : A -> Prop) : Set :=
| Unknown : maybe P
| Found : forall x : A, P x -> maybe P.
```

Using `maybe`, we can define a certified predecessor function as:

```
Definition predcert : forall n : nat, {{m | n = S m}}.
  refine (fun n =>
    match n return {{m | n = S m}} with
      | O => ??
      | S n' => [ n' ]
    end); trivial.
Defined.
```

The previous definition uses some notations: first, type `maybe P` is denoted by `{x | P}`. Constructor `Unknown` is represented by `??` and `Found n` by `[ n ]`. In our development, we use these specification types to define several certified functions. More details about these will be given in Section 3.2.3.

A detailed discussion on using Coq is out of the scope of this paper. Good introductions to Coq proof assistant are available elsewhere [7, 10].

3. Proposed Semantic

This chapter presents the two operational semantic we propose in this work. The first one is the small-step operational semantics. This version does not deal with problematic REs. Later, we propose a big-step operational semantics, which can deal correctly with problematic REs and was our basis for the Coq formalization of our algorithm.

3.1. Small-step Operational Semantics

In this section, we present the definition of an operational semantics for RE parsing which is equivalent to executing the Thompson's construction NFA over the input string. Observe that the inductive semantics for RE (Figure 2.2) can be understood as a big-step operational semantics for RE, since it ignores many details on how should we proceed to match an input [46].

The semantics is defined as a binary relation between *configurations*, which are 5-uples $\langle d, e, c, b, s\rangle$ where:

- d is a direction, which specifies if the semantics is starting (denoted by B) or finishing (F) the processing of the current expression e.
- e is the current expression being evaluated;
- c is a context in which e occurs. Contexts are just a list of Hole type (defined in Section 2.6.2) in our implementation.
- b is a bit-code for the current parsing result, in reverse order.
- s is the input string currently being processed.

Notation $\langle d, e, c, b, s\rangle \rightarrow \langle d', e', c', b', s'\rangle$ denotes that from configuration $\langle d, e, c, b, s\rangle$ we can give a step to $\langle d', e', c', b', s'\rangle$ using the rules specified in Figure 3.1.

The semantics rules can be divided in two groups: starting rules and finishing rules. Starting rules deal with configurations with a begin (B) direction and denote that we are beginning the parsing for its RE e. Finishing rules use the context to decide how the parsing for some expression should end. Intuitively, starting rules correspond to transitions entering a sub-automaton of Thompson NFA and finishing rules to transitions exiting a sub-automaton.

The meaning of each starting rule is as follows. Rule $\{Eps\}$ specifies that we can mark a state as finished if it consists of a starting configuration with RE ϵ. We can finish any configuration for RE Chr a if the current string starts with a leading a. Whenever we

$$\frac{}{\langle B, \epsilon, c, b, s\rangle \rightarrow \langle F, \epsilon, c, b, s\rangle}\ (Eps) \qquad \frac{}{\langle B, a, c, b, a : s\rangle \rightarrow \langle F, a, c, b, s\rangle}\ (Chr)$$

$$\frac{c' = E[\,] + e' : c}{\langle B, e + e', c, b, s\rangle \rightarrow \langle B, e, c', b', s\rangle}\ (Left_B) \qquad \frac{c' = e + E[\,] : c}{\langle B, e + e', c, b, s\rangle \rightarrow \langle B, e', c', b', s\rangle}\ (Right_B)$$

$$\frac{c' = E[\,]e' : c}{\langle B, ee', c, b, s\rangle \rightarrow \langle B, e, c', b, s\rangle}\ (Cat_B) \qquad \frac{}{\langle B, e^{\star}, c, b, s\rangle \rightarrow \langle B, e, \star : c, \mathtt{0_b} : b, s\rangle}\ (Star_1)$$

$$\frac{}{\langle B, e^{\star}, c, b, s\rangle \rightarrow \langle F, e^{\star}, c, \mathtt{1_b} : b, s\rangle}\ (Star_2) \qquad \frac{c' = eE[\,] : c}{\langle F, e, E[\,]e' : c, b, s\rangle \rightarrow \langle B, e', c', b, s\rangle}\ (Cat_{EL})$$

$$\frac{}{\langle F, e', eE[\,] : c, b, s\rangle \rightarrow \langle F, ee', c, b, s\rangle}\ (Cat_{ER}) \qquad \frac{c = E[\,] + e' : c'}{\langle F, e, c, b, s\rangle \rightarrow \langle F, e + e', c', \mathtt{0_b} : b, s\rangle}\ (Left_E)$$

$$\frac{c = e + E[\,] : c'}{\langle F, e', c, b, s\rangle \rightarrow \langle F, e + e', c', \mathtt{1_b} : b, s\rangle}\ (Right_E) \qquad \frac{}{\langle F, e, \star : c, b, s\rangle \rightarrow \langle B, e, \star : c, \mathtt{0_b} : b, s\rangle}\ (Star_{E1})$$

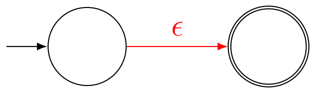
$$\frac{}{\langle F, e, \star : c, b, s\rangle \rightarrow \langle F, e^{\star}, c, \mathtt{1_b} : b, s\rangle}\ (Star_{E2})$$

Figure 3.1.: Small-step semantics for RE parsing.

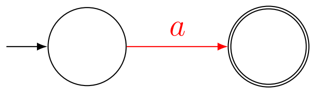
have a starting configuration with a choice RE, $e_1 + e_2$, we can non-deterministically choose if the input string s can be processed by e_1 (rule $Left_B$) or e_2 (rule $Right_B$). For beginning configurations with concatenation, we parse the input string using each of its components sequentially. Finally, for starting configurations with a Kleene star operator, $e^{\star}$, we can either start the processing of e or finish the processing for $e^{\star}$. In all recursive cases for RE, we insert context information in the third component of the resulting configuration in order to decide how the machine should step after finishing the execution of the RE currently on focus.

Rule (Cat_{EL}) applies to any configuration which is finishing with a left concatenation context $(E[\,]e')$. In such situation, such rule specifies that a computation should continue with e' and push the context $e\,E[\,]$. We end the computation for a concatenation whenever we find a context $e\,E[\,]$ in the context component (rule (Cat_{ER})). Finishing a computation for choice consists in just popping its correspondent context, as done by rules $(Left_E)$ and $(Right_E)$. For the Kleene star operator, we can either finish the computation by popping the contexts and adding the corresponding $\mathtt{1_b}$ to end its matching list or restart with RE e for another matching over the input string.

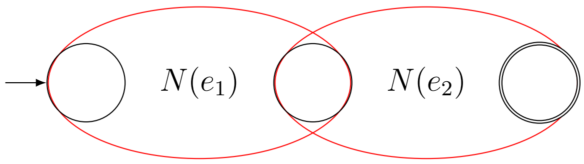
The proposed semantics is inspired by Thompson's NFA construction (as shown in Section 2.3). First, the rule *Eps* can be understood as executing the transition highlighted in red in the following schematic automaton.



The *Chr* rule corresponds to the following transition (represented in red) in the next automaton.

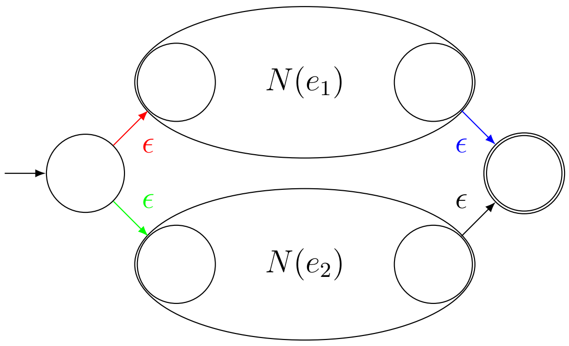


Rule Cat_B corresponds to start processing the input string in the automaton $N(e_1)$; while rule Cat_{EL} deals with exiting the automaton $N(e_1)$ followed by processing the remaining string in $N(e_2)$. Rule Cat_{ER} deals with ending the processing in the automaton below.

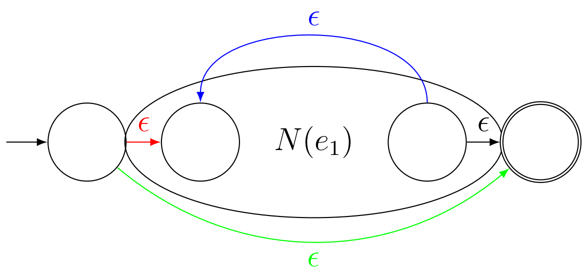


If we consider a RE $e = e_1 + e_2$ and let $N(e_1)$ and $N(e_2)$ be two NFAs for e_1 and e_2, respectively, we have the following correspondence between transitions and semantics rules in the next NFA:

- Red transition for rule $Left_B$;
- Green for $Right_B$;
- Blue for $Left_E$; and
- Black for $Right_E$.



Finally, we present Kleene star rules in next automaton according to Thompson's NFA construction. The colors are red for $Star_1$ rule, green for $Star_2$, blue for $Star_{E1}$ and black for $Star_{E2}$.



The starting state of the semantics is given by the configuration $\langle B, e, [], [], s\rangle$ and the accepting configurations are $\langle F, e', [], bs, []\rangle$, for some RE e' and code bs. Following common practice, we let $\rightarrow^\star$ denote the reflexive, transitive closure of the small-step semantics defined in Figure 3.1. We say that a string s is accepted by RE e if $\langle B, e, [], [], s\rangle \rightarrow^\star \langle F, e, [], bs, []\rangle$. The next theorem asserts that our semantics is sound and complete with respect to RE inductive semantics (Figure 2.2).

Theorem 8. *For all strings s and non-problematic REs e, $s \in [\![e]\!]$ if, and only if, $\langle B, e, [], [], s\rangle \rightarrow^\star \langle F, e, [], b, []\rangle$ and $\langle F, e, [], b, []\rangle$ is an accepting configuration.*

Proof. ($\rightarrow$): We proceed by induction on the derivation of $s \in [\![e]\!]$.

1. Case rule *Eps*: Then, $e = \epsilon$, $s = \epsilon$ and the conclusion is immediate.
2. Case rule *Chr*: Then, $e = a$, $s = a$ and the conclusion follows.
3. Case rule *Left*: Then, $e = e_1 + e_2$ and $s \in [\![e_1]\!]$. By the induction hypothesis, we have $\langle B, e_1, ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$ and the conclusion follows.
4. Case rule *Right*: Then, $e = e_1 + e_2$ and $s \in [\![e_2]\!]$. By the induction hypothesis, we have $\langle B, e_2, ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$ and the conclusion follows.
5. Case rule *Cat*: Then, $e = e_1 e_2$, $s_1 \in [\![e_1]\!]$, $s_2 \in [\![e_2]\!]$ and $s = s_1 s_2$. By the induction hypothesis on $s_1 \in [\![e_1]\!]$ we have that $\langle B, e_1, ctx, b, s\rangle \rightarrow^\star \langle F, e, E[\,]\,e_2 : ctx, b', []\rangle$ and by induction hypothesis on $s_2 \in [\![e_2]\!]$, we have $\langle B, e_2, e_1\,E[\,] : ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx, b', []\rangle$ and the conclusion follows.
6. Case rule *StarBase*: Then, $e = e_1^\star$ and $s = \epsilon$. The conclusion is immediate.
7. Case rule *StarRec*: Then, $e = e_1^\star$, $s = s_1 s_2$, $s_1 \in [\![e_1]\!]$ and $s_2 \in [\![e_1^\star]\!]$. By the induction hypothesis on $s_1 \in [\![e_1]\!]$, we have $\langle B, e_1, \star : ctx, b, s_1\rangle \rightarrow^\star \langle F, e, \star : ctx, b', []\rangle$, the induction hypothesis on $s_2 \in [\![e_1^\star]\!]$ give us $\langle B, e_1^\star, \star : ctx, b, s_2\rangle \rightarrow^\star \langle F, e, \star : ctx, b', []\rangle$ and conclusion follows.

($\leftarrow$): We proceed by induction on e.

1. Case $e = \epsilon$. Then, we have $\langle B, \epsilon, ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$ and $s = \epsilon$. Conclusion follows by rule *Eps*.

2. Case $e = a$. Then $\langle B, a, ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$ and $s = a$. Conclusion follows by rule Chr.

3. Case $e = e_1 + e_2$. Now, we consider the following cases.

   a) s is accepted by e_1. Then, we have the following derivation:

   $$\langle B, e_1 + e_2, ctx, b, s\rangle \rightarrow \langle B, e_1, E[\,] + e_2 : ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$$

   By induction hypothesis on e_1 and the derivation $\langle B, e_1, E[\,]+e_2 : ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$ we have $s \in [\![e_1]\!]$ and the conclusion follows by rule $Left$.

   b) s is accepted by e_2. Then, we have the following derivation:

   $$\langle B, e_2, ctx, b, s\rangle \rightarrow \langle B, e_1, e_1 + E[\,] : ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$$

   By induction hypothesis on e_2 and the derivation $\langle B, e_1, e_1+E[\,] : ctx, b, s\rangle \rightarrow^\star \langle F, e, ctx', b', []\rangle$, we have $s \in [\![e_2]\!]$ and conclusion follows by rule $Right$.

□

3.1.1. Implementation Details

We chose Haskell to implement the first version of our VM-based algorithm due to Haskell's easiness on quickly prototyping an interpreter for our small-step semantics. Thus, it could be easier and faster to discover errors in our semantics formulation, mainly because of QuickCheck (Section 2.6.1).

In order to implement the small-step semantics of Figure 3.1, we need to represent configurations. We use type Conf to denote configurations and directions are represented by type Dir, where Begin denote the starting and End the finishing direction.

```
data Dir = Begin | End
type Conf = (Dir, Regex, [Hole], Code, String)
```

Function finish tests if a configuration is an accepting one.

```
finish :: Conf → Bool
finish (End, _, [ ], _, [ ]) = True
finish _ = False
```

The small-step semantics is implemented by function next, which returns a list of configurations that can be reached from a given input configuration. We will begin by explaining the equations that code the set of starting rules from the small-step semantics. The first alternative

```
next :: Conf → [Conf]
next (Begin, ϵ, ctx, bs, s) = [(End, ϵ, ctx, bs, s)]
```

implements rule (*Eps*), which finishes a starting Conf with an ϵ. Rule (*Chr*) is implemented by the following equation

```
next (Begin, Chr c, ctx, bs, a : s)
    | a ≡ c = [(End, Chr c, ctx, bs, s)]
    | otherwise = []
```

which consumes an input character a if it matches RE Chr c; otherwise it fails by returning an empty list. For a choice expression, we can use two distinct rules: one for parsing the input using its left component and another rule for the right one. Since both union and Kleene star introduce non-determinism in RE parsing, we can easily model this using the list monad, by return a list of possible resulting configurations.

```
next (Begin, e + e′, ctx, bs, s)
    = [(Begin, e, InChoiceL e′ : ctx, 0_b : bs, s)
      , (Begin, e′, InChoiceR e : ctx, 1_b : bs, s)]
```

Concatenation just sequences the computation of each side of its composing RE.

```
next (Begin, e • e′, ctx, bs, s)
    = [(Begin, e, InCatL e′ : ctx, bs, s)]
```

For a starting configuration with Kleene star operator, Star e, we can proceed in two ways: by beginning the parsing of RE e or by finishing the computation for Star e over the input.

```
next (Begin, Star e, ctx, bs, s)
    = [(Begin, e, InStar : ctx, 0_b : bs, s)
      , (End, (Star e), ctx, 1_b : bs, s)]
```

The remaining equations of next deal with operational semantics finishing rules. The equation below implements rule (Cat_{EL}) which specifies that an ended computation for the left component of a concatenation should continue with its right component.

```
next (End, e, InCatL e′ : ctx, bs, s)
    = [(Begin, e′, InCatR e : ctx, bs, s)]
```

Whenever we are in a finishing configuration with a right concatenation context, (InCatR e), we end the parsing of the input for the whole concatenation RE.

```
next (End, e′, InCatR e : ctx, bs, s)
    = [(End, e • e′, ctx, bs, s)]
```

Next equations implement the rules that finish configurations for the union, by committing to its first successful branch.

```
next (End, e, InChoiceL e′ : ctx, bs, s)
    = [(End, e + e′, ctx, 0_b : bs, s)]
```

```
next (End, e′, InChoiceR e : ctx, bs, s)
   = [(End, e + e′, ctx, 1_b : bs, s)]
```

Equations for Kleene star implement rules ($Star_{E1}$) and ($Star_{E2}$) which allows ending or add one more match for an RE e.

```
next (End, e, InStar : ctx, bs, s)
   = [(Begin, e, InStar : ctx, 0_b : bs, s)
     , (End, (Star e), ctx, 1_b : bs, s)]
```

Finally, stuck states on the semantics are properly handled by the following equation which turns them all into a failure (empty list).

```
next _ = [ ]
```

The reflexive-transitive closure of the semantics is implemented by function steps, which computes the trace of all states needed to determine if a string can be parsed by the RE e.

```
steps :: [Conf] → [Conf]
steps [ ] = [ ]
steps cs = steps [c′ | c ← cs, c′ ← next c] ++ cs
```

Finally, the function for parsing a string using an input RE is implemented as follow s:

```
vmAccept :: String → Regex → (Bool, Code)
vmAccept s e = let r = [c | c ← steps init_cfg, finish c]
  in if null r then (False, [ ]) else (True, bitcode (head r))
  where
    init_cfg = [(Begin, e, [ ], [ ], s)]
    bitcode (_, _, _, bs, _) = reverse bs
```

Function vmAccept returns a pair formed by a boolean and the bit-code produced during the parsing of an input string and RE. Observe that we need to reverse the bit-codes, since they are built in reverse order.

3.1.2. Experiments

Test case generators. In order to test the correctness of our semantics, we needed to build generators for REs and strings. We used the QuickCheck library to develop functions to randomly generate strings accepted and rejected for a RE.

Generation of random REs is done by function sizedRegex, which takes a depth limit to restrict the size of the generated RE. Whenever the input depth limit is less or equal to 1, we can only build a ϵ or a single character RE. The definition of sizedRegex uses QuickCheck function frequency, which receives a list of pairs formed by a weight and a random generator and produces, as result, a generator which uses such frequency distribution. In sizedRegex implementation we gave a higher weight to generate characters and equal distributions to build concatenation, union or star.

```
sizedRegex :: Int → Gen Regex
sizedRegex n
  | n ⩽ 1 = frequency [(10, return ϵ), (90, Chr ⟨$⟩ genChar)]
  | otherwise = frequency [(10, return ϵ), (30, Chr ⟨$⟩ genChar)
    , (20, ( • ) ⟨$⟩ sizedRegex n2 ⟨⋆⟩ sizedRegex n2)
    , (20, ( + ) ⟨$⟩ sizedRegex n2 ⟨⋆⟩ sizedRegex n2)
    , (20, Star ⟨$⟩ suchThat (sizedRegex n2) (not ∘ nullable))]
    where n2 = div n 2
```

For simplicity and brevity, we only generated REs that do not contain sub-REs of the form $e^\star$, where e is nullable[1]. All results can be extended to problematic[2] REs in the style of Frisch et. al [21].

Given an RE e, we can generate a random string s such that $s \in [\![e]\!]$ using next definition. We generate strings by choosing randomly between branches of a union or by repeating n times a string s which is accepted by e, whenever we have $e^\star$ (function randomMatches).

```
randomMatch :: Regex → Gen String
randomMatch ϵ = return ""
randomMatch (Chr c) = return [c]
randomMatch (e • e′) = liftM2 (++) (randomMatch e)
  (randomMatch e′)
randomMatch (e + e′) = oneof [randomMatch e, randomMatch e′]
randomMatch (Star e) = do
  n ← choose (0, 3) :: Gen Int
  randomMatches n e

randomMatches :: Int → Regex → Gen String
randomMatches m e′
  | m ⩽ 0 = return []
  | otherwise = liftM2 (++) (randomMatch e′)
    (randomMatches (m − 1) e′)
```

The algorithm for generating random strings that aren't accepted by a RE is similarly defined.

Properties considered. In order to verify if the defined semantics is correct, we need to check the following properties:

[1] A RE e is *nullable* if $\epsilon \in [\![e]\!]$.

[2] We say that a RE e is problematic if there's e' such that $e = e'^\star$ and $\epsilon \in [\![e']\!]$.

1. Our semantics accepts only and all the strings in the language described by the input RE: we test this property by generating random strings that should be accepted and strings that must be rejected by a random RE.

2. Our semantics generates valid parsing evidence: the bit-codes produced as result have the following properties: 1) the bit-codes can be parsed into a valid parse tree t for the random produced RE e, i.e. $\vdash t : e$ holds ; 2) flat t = s and 3) code e t = bs.

Note that we need a correct implementation of RE parsing to verify the first property. For this, we used the accept function from [19] and compared its result with vmAccept's. The second property demands that the bit-codes produced can be decoded into valid parsing evidence. The verification of produced bit-codes is done by function validCode shown below.

```
validCode :: String → Code → Regex → Bool
validCode _ [] _ = True
validCode s bs e = case decode e bs of
  Just t → and [tc t e, flat t ≡ s, code t e ≡ bs]
  _ → False
```

Finally, function vmCorrect combines both properties mentioned above into a function that is called to test the semantics implementation.

```
vmCorrect :: Regex → String → Property
vmCorrect e s
   = let (r, bs) = vmAccept s e
     in (accept e s ≡ r) ∧ validCode s bs e
```

In addition to coding / decoding of parse trees, we need a function which checks if a tree is indeed a parsing evidence for some RE e. Function tc takes, as arguments, a parse tree t and a RE e and verifies if t is an evidence for e.

```
tc :: Tree → Regex → Bool
tc () ϵ = True
tc (Chr c) (Chr c′) = c ≡ c′
tc (t • t′) (e • e′) = tc t e ∧ tc t′ e′
tc (InL t) (e + _) = tc t e
tc (InR t′) (_ + e′) = tc t′ e′
tc (List ts) (Star e) = all (flip tc e) ts
```

Function tc is a implementation of parsing tree typing relation, as specified by the following result.

Theorem 9. *For all tree t and RE e, $\vdash t : e$ if, and only if,* tc t e = True.

Proof. ($\rightarrow$): We proceed by induction on the derivation of $\vdash t : e$.

1. Case rule $T1$: Then, $e = \epsilon$ and $\mathsf{t} = ()$ and conclusion follows.
2. Case rule $T2$: Then, $e = a$ and $\mathsf{t} =$ Chr a and conclusion follows.
3. Case rule $T3$: Then, $e = e_1 + e_2$ and $\mathsf{t} =$ InL tl, where $\vdash tl : e_1$. By induction hypothesis, we have that tc tl e1 $=$ True and conclusion follows.
4. Case rule $T4$: Then, $e = e_1 + e_2$ and $\mathsf{t} =$ InR tr, where $\vdash tr : e_2$. By induction hypothesis, we have that tc tr e2 $=$ True and conclusion follows.
5. Case rule $T5$: Then, $e = e_1\, e_2$ and $\mathsf{t} =$ tl • tr. Conclusion is immediate from the induction hypothesis.
6. Case rule $T6$: Then, $e = e_1^\star$ and $\mathsf{t} =$ List ts and conclusion follows from the induction hypothesis on each element of ts.

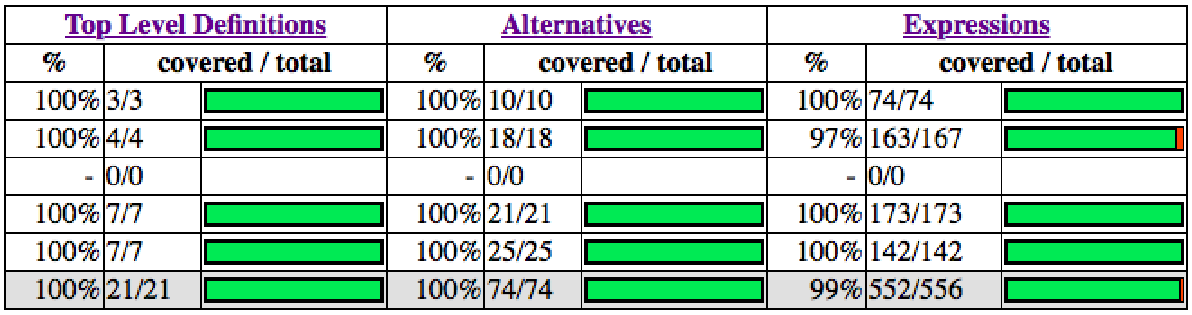
($\leftarrow$): We proceed by induction on e.

1. Case $e = \epsilon$: Then, $\mathsf{t} = ()$ and the conclusions follows by rule $T1$.
2. Case $e = a$: Then, $\mathsf{t} =$ Chr a and the conclusions follows by rule $T2$.
3. Case $e = e_1 + e_2$: Now, we consider the following subcases:
   a) Case $\mathsf{t} =$ InL tl: By induction hypothesis, we have that tc tl e1 $=$ True and conclusion follows.
   b) Case $\mathsf{t} =$ InR tr: By induction hypothesis, we have that tc tr e2 $=$ True and conclusion follows.
4. Case $e = e_1\, e_2$: Then, $\mathsf{t} =$ tl • tr and conclusion follows by the induction hypothesis and the rule $T5$.
5. Case $e = e_1^\star$: Then, $\mathsf{t} =$ List ts and conclusion follows by induction hypothesis on each element of ts and rule $T6$.

□

Code coverage results. After running thousands of well-succeeded tests, we gain a high degree of confidence in the correctness of our semantics. However, it is important to measure how much of our code is covered by the test suite. We use the Haskell Program Coverage tool (HPC) [23] to generate statistics about the execution of our tests. Code coverage results are presented in Figure 3.2.

Our test suite gave us almost 100% of code coverage, which provides a strong evidence that our small-step semantics is indeed correct. All top level definitions and function alternatives are actually executed by the test cases and just two expressions are marked as non-executed by HPC.



Top Level Definitions			Alternatives			Expressions		
%	covered / total		%	covered / total		%	covered / total	
100%	3/3	🟩	100%	10/10	🟩	100%	74/74	🟩
100%	4/4	🟩	100%	18/18	🟩	97%	163/167	🟩🟥
-	0/0		-	0/0		-	0/0	
100%	7/7	🟩	100%	21/21	🟩	100%	173/173	🟩
100%	7/7	🟩	100%	25/25	🟩	100%	142/142	🟩
100%	21/21	🟩	100%	74/74	🟩	99%	552/556	🟩🟥

Figure 3.2.: Code coverage results

3.2. Big-step Operational Semantics

The small-step semantics presented in Section 3.1 was our first attempt to develop a VM-based algorithm for the RE parsing problem. Despite its high coverage results when submitted to QuickCheck, that semantics has some issues. As we stated previously, it does not work with problematic REs.

To solve the first problem, we adopted a function which converts a problematic RE into an equivalent non-problematic one, as proposed by Medeiros et al. [41]. In order to formalize our small-step operational semantics in Coq, we now propose a big-step one for it, which is easier to understand and formalize in Coq and behaves the same way as the small-step one. In fact, we consider the small-step version as an intermediate step to achieve the big-step one presented in this section.

3.2.1. Dealing With Problematic REs

A known problem in RE parsing is how to deal with the so-called problematic REs. A naive approach for parsing problematic REs can make the algorithm loop [21]. Medeiros et al. [41] present a function which converts a problematic RE into a equivalent non-problematic one.

The conversion function relies on two auxiliary definitions: one for testing if a RE accepts the empty string and another to test if a RE is equivalent to ϵ. We name such functions as `nullable` and `empty`, respectively.

$$
\begin{array}{lcl}
\texttt{nullable}(\emptyset) & = & \bot \\
\texttt{nullable}(\epsilon) & = & \top \\
\texttt{nullable}(a) & = & \bot \\
\texttt{nullable}(e_1 + e_2) & = & \texttt{nullable}(e_1) \lor \texttt{nullable}(e_2) \\
\texttt{nullable}(e_1\, e_2) & = & \texttt{nullable}(e_1) \land \texttt{nullable}(e_2) \\
\texttt{nullable}(e^\star) & = & \top \\
\\
\texttt{empty}(\emptyset) & = & \bot \\
\texttt{empty}(\epsilon) & = & \top \\
\texttt{empty}(a) & = & \bot \\
\texttt{empty}(e_1 + e_2) & = & \texttt{empty}(e_1) \land \texttt{empty}(e_2) \\
\texttt{empty}(e_1\, e_2) & = & \texttt{empty}(e_1) \land \texttt{empty}(e_2) \\
\texttt{empty}(e^\star) & = & \texttt{empty}(e)
\end{array}
$$

Functions `nullable` and `empty` obey the following correctness properties.

Lemma 1. *$\texttt{nullable}(e) = \top$ if, and only if, $\epsilon \in [\![e]\!]$.*

Proof.
($\rightarrow$) Induction over the structure of e.
($\leftarrow$) Induction over the derivation of $\epsilon \in [\![e]\!]$. □

Lemma 2. *If $\texttt{empty}(e) = \top$ then $e \approx \epsilon$.*

Proof. Induction over the structure of e. □

Given these two predicates, Medeiros et.al. [41] define two mutually recursive functions, named $\texttt{f}_{\texttt{in}}$ and $\texttt{f}_{\texttt{out}}$. The function $\texttt{f}_{\texttt{out}}$ recurses over the structure of an input RE searching for a problematic subexpression and $\texttt{f}_{\texttt{in}}$ rewrites the Kleene star subexpression so that it becomes non-problematic and preserves the original RE language. The definition of functions $\texttt{f}_{\texttt{in}}$ and $\texttt{f}_{\texttt{out}}$ are presented next.

$$
\begin{array}{lcl}
\texttt{f}_{\texttt{out}}(e) & = & e,\ \texttt{if}\ \ e = \epsilon, e = \emptyset\ \texttt{ or }\ e = a \\
\texttt{f}_{\texttt{out}}(e_1 + e_2) & = & \texttt{f}_{\texttt{out}}(e_1) + \texttt{f}_{\texttt{out}}(e_2) \\
\texttt{f}_{\texttt{out}}(e_1\, e_2) & = & \texttt{f}_{\texttt{out}}(e_1)\, \texttt{f}_{\texttt{out}}(e_2) \\
\texttt{f}_{\texttt{out}}(e^\star) & = & \begin{cases} \texttt{f}_{\texttt{out}}(e)^\star & \texttt{if}\ \neg\,\texttt{nullable}(e) \\ \epsilon & \texttt{if empty}(e) \\ \texttt{f}_{\texttt{in}}(e)^\star & \texttt{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{f_{in}}(e_1\,e_2) & = & \mathtt{f_{in}}(e_1 + e_2) \\
\mathtt{f_{in}}(e_1 + e_2) & = & \begin{cases}
\mathtt{f_{in}}(e_2) & \texttt{if } \mathtt{empty}(e_1) \land \mathtt{nullable}(e_2) \\
\mathtt{f_{out}}(e_2) & \texttt{if } \mathtt{empty}(e_1) \land \neg\mathtt{nullable}(e_2) \\
\mathtt{f_{in}}(e_1) & \texttt{if } \mathtt{nullable}(e_1) \land \mathtt{empty}(e_2) \\
\mathtt{f_{out}}(e_1) & \texttt{if } \neg\mathtt{nullable}(e_1) \land \mathtt{empty}(e_2) \\
\mathtt{f_{out}}(e_1) + \mathtt{f_{in}}(e_2) & \texttt{if } \neg\mathtt{nullable}(e_1) \land \neg\mathtt{empty}(e_2) \\
\mathtt{f_{in}}(e_1) + \mathtt{f_{out}}(e_2) & \texttt{if } \neg\mathtt{empty}(e_1) \land \neg\mathtt{nullable}(e_2) \\
\mathtt{f_{in}}(e_1) + \mathtt{f_{in}}(e_2) & \texttt{otherwise}
\end{cases} \\
\mathtt{f_{in}}(e^\star) & = & \begin{cases}
\mathtt{f_{in}}(e) & \texttt{if } \mathtt{nullable}(e) \\
\mathtt{f_{out}}(e) & \texttt{otherwise}
\end{cases}
\end{array}
$$

The result of applying $\mathtt{f_{out}}$ on a RE is producing an equivalent non-problematic one. This fact is expressed by the following theorem.

Theorem 10. *If $f_{out}(e) = e'$ then $e \approx e'$ and e' is a non-problematic RE.*

Proof. Well-founded induction on the complexity of (e, s), where s is an arbitrary string, using several lemmas about RE equivalence and lemmas 1 and 2. □

This result is proved (informally[3]) by Medeiros et. al. [41]. In order to formalize this result in Coq, we needed to prove several theorems about RE equivalence. We postpone the discussion on some details of our formalization to Section 3.2.3.

3.2.2. Big-step Operational Semantics

We now present the definition of a big step operational semantics for a RE parsing VM. The state of our VM is a pair formed by the current RE and the string under parsing. Each machine transition may produce, as a side effect, a bit-coded parse tree and the remaining string to be parsed. We denote our semantics by a judgment of the form $\langle e, s\rangle \rightsquigarrow (bs, s_p, s_r)$, where e is current RE, s is the input string, bs is the produced bit-coded tree, s_p is the parsed prefix of the input string and s_r is the yet to be parsed string. We let notation $\langle e, s\rangle \not\rightsquigarrow$ denote the fact that string s cannot be parsed by RE e.

The meaning of each semantics rules is as follows. Rule $EpsVM$ specifies that parsing s using RE ϵ produces an empty list of bits and does not consume any symbol from s. Rule $ChrVM$ consumes the first symbol of the input string if it matches the input RE. Rules $LeftVM$ and $RightVM$ specifies how the semantics executes an RE $e + e'$, by trying to parse the input using either the left or right subexpression. Note that, as a result, we append a bit 0_b when we successfully parse the input using the left choice operand and the bit 1_b for a parsing using the right operand. Rule $CatVM$ defines how a concatenation $e_1\,e_2$ is executed by the semantics: first, the input is parsed using the RE e_1 and the remaining string is used as input to execute e_2. The bit-coded tree for the $e_1\,e_2$ is just the concatenation of the produced codes for e_1 and e_2. Rules $NilVM$ and

[3]By "informally", we mean that the result is not mechanized in a proof assistant.

$$\frac{}{\langle \epsilon, s\rangle \leadsto ([\,], \epsilon, s)}\{EpsVM\} \qquad \frac{}{\langle a, as\rangle \leadsto ([\,], a, s)}\{ChrVM\}$$

$$\frac{\langle e_1, s\rangle \leadsto (b, s_p, s_r)}{\langle e_1 + e_2, s\rangle \leadsto (0_b\, b, s_p, s_r)}\{LeftVM\} \qquad \frac{\langle e_2, s\rangle \leadsto (b, s_p, s_r)}{\langle e_1 + e_2, s\rangle \leadsto (1_b\, b, s_p, s_r)}\{RightVM\}$$

$$\frac{\begin{array}{c}\langle e_1, s\rangle \leadsto (b_1, s_{p1}, s_1)\\ \langle e_2, s_1\rangle \leadsto (b_2, s_{p2}, s_r)\end{array}}{\langle e_1\, e_2, s\rangle \leadsto (b_1\, b_2, s_{p1}\, s_{p2}, s_r)}\{CatVM\} \qquad \frac{\langle e, s\rangle \not\leadsto}{\langle e^\star, s\rangle \leadsto (1_b, \epsilon, s)}\{NilVM\}$$

$$\frac{\langle e, s\rangle \leadsto (b_1, s_{p1}, s_1) \quad s_{p1} \neq \epsilon \quad \langle e^\star, s_1\rangle \leadsto (b_2, s_{p2}, s_r)}{\langle e^\star, s\rangle \leadsto (0_b\, b_1\, b_2, s_{p1}\, s_{p2}, s_r)}\{ConsVM\}$$

Figure 3.3.: Big-step operational semantics for RE parsing.

$ConsVM$ deal with unproblematic Kleene star REs. The rule $NilVM$ is only applicable when it is not possible to parse the input using the RE e in $e^\star$. Rule $ConsVM$ can be used whenever we can parse the input using e and the parsed prefix is not an empty string. The remaining string (s_1) of e's parsing is used as input for the next iteration of RE $e^\star$ parsing.

Evidently, the proposed semantics is sound and complete w.r.t. standard RE semantics and only produces valid parsing evidence.

Theorem 11 (Soundness). *If $\langle e, s\rangle \leadsto (bs, s_p, s_r)$ then $s = s_p\, s_r$ and $s_p \in [\![e]\!]$.*

Proof. Well-founded induction on the complexity of (e, s). □

Theorem 12 (Completeness). *If $s_p \in [\![e]\!]$ then for all s_r we have that exists bs, s.t. $\langle e, s_p\, s_r\rangle \leadsto (bs, s_p, s_r)$.*

Proof. Well-founded induction on the complexity of (e, s). □

Theorem 13 (Parsing result soundness). *If $\langle e, s\rangle \leadsto (bs, s_p, s_r)$ then: 1) $bs \rhd e$; 2) $\mathtt{flatten}(\mathtt{decode}(bs : e)) = s_p$; and 3) $\mathtt{code}(\mathtt{decode}(bs : e) : e) = bs$.*

Proof. Well-founded induction on the complexity of (e, s) using Theorem 3. □

3.2.3. Coq Formalization

In this subsection, we describe the main design decisions in our formalization. At the end of this subsection, we discuss how we extract a Haskell implementation from our Coq development.

RE syntax and semantics Our representation of RE syntax and semantics is as usual in type theory-based proof assistants. We use an inductive type to represent RE syntax and an inductive predicate to denote its semantics.

```
Inductive regex : Set :=
| Empty : regex | Eps : regex | Chr : ascii -> regex
| Cat : regex -> regex -> regex
| Choice : regex -> regex -> regex
| Star   : regex -> regex.
```

Type `regex` represents RE syntax and its definition is straightforward. We use some notations to write `regex` values. We let `#0` denote `Empty`, `#1` represents `Eps`, while infix operators `:+:` and `@` denote `Choice` and `Cat`. Finally, `Star e` is written `(e ^*)`.

RE semantics is represented by type `in_regex` which has a constructor for each rule of the semantics presented in Figure 2.2.

```
Inductive in_regex : string -> regex -> Prop :=
| InEps : "" <<- #1
| InChr : forall c, String c "" <<- ($ c)
| InLeft
  :  forall s e e'
  ,  s <<- e
  -> s <<- (e :+: e')
| InStarRight
  : forall s s' e s1
  , s <> ""
  -> s <<- e
  -> s' <<- (e ^*)
  -> s1 = s ++ s'
  -> s1 <<- (e ^*)
...  (** some constructors omitted. *)
where "s '<<-' e" := (in_regex s e).
```

We use notation `s <<- e` to denote `in_regex s e`.

RE equivalence Using the previous presented semantics, we can define RE equivalence by coding its standard definition in Coq as:

```
Definition regex_equiv (e e' : regex) : Prop :=
  forall s, s <<- e <-> s <<- e'.
```

We use notation `e1 === e2` to denote `regex_equiv e1 e2`. In our formalization, we proved that `regex_equiv` is an equivalence relation, which is necessary to allow the rewriting of such equalities by Coq tactics.

In order to complete our formalization, we needed several results about RE equivalence. Most of them are proved by well-founded induction on the complexity of a pair formed by a RE and a string (defined in Section 2.2). In order to formalize the needed ordering relation, we take advantage of Coq's standard library, which provides several combinators to assemble well-founded relations. As an example, consider the following fact used by Medeiros et. al [41] to prove the correctness of its $\texttt{f}_{\texttt{out}}$ function: $(e_1 + e_2)^\star \approx (e_1\, e_2)^\star$, which holds if both e_1 and e_2 accepts the empty string. In our formalization, such equivalence is proved by the following theorem by well-founded induction.

```
Lemma choice_star_cat_star
   : forall e1 e2, "" <<- e1 -> "" <<- e2 ->
                    ((e1 @ e2) ^*) === ((e1 :+: e2) ^*).
```

Several other lemmas about RE equivalence were proved in order to complete the formalization of the problematic RE conversion function. We omit them for brevity.

Converting problematic REs The first step to certify the algorithm for converting problematic REs into non-problematic ones is to define the predicates for testing whether an input RE is nullable or equivalent to ϵ. We define such functions using dependently typed programming, i.e. its types provide certificates that the result has its desired correctness property.

The nullability test is represented by function `null`:

```
Definition null : forall e, {"" <<- e} + {~ "" <<- e}.
  refine (fix null e : {"" <<- e} + {~ "" <<- e} :=
            match e as e' return e = e' ->
                    {"" <<- e'} + {~ "" <<- e'} with
            | #1 => fun Heq => Yes
            | e1 @ e2 => fun Heq =>
               match null e1 , null e2 with
               | Yes , Yes  => Yes
               | _ , _    => No
               end
            | e1 :+: e2 => fun Heq => ...
            | e1 ^* => fun Heq => Yes
            end (eq_refl e)) ...
           (** some cases and tactics omitted *)
```

Its type specifies that, for any RE e, either e accepts the empty string (i.e. `"" <<- e` holds) or not (`~ "" <<- e`). Since such function contains proofs terms, we use tactic `refine` to define its computation content leaving the logical subterms to be filled by tactics. The definition of `null` employs the convoy-pattern [10], which consists in introducing an equality to allow the refinement of each equation type in dependently typed pattern-matching.

In order to specify the emptiness test predicate, we use an inductive type which characterizes when a RE is equivalent to ϵ.

```
Inductive empty_regex : regex -> Prop :=
| Emp_Eps : empty_regex #1
| Emp_Cat : forall e e', empty_regex e ->
                    empty_regex e' ->
                    empty_regex (e @ e')
| Emp_Choice : forall e e', empty_regex e ->
                    empty_regex e' ->
                    empty_regex (e :+: e')
| Emp_Star : forall e, empty_regex e ->
                  empty_regex (e ^*).
```

The meaning of each constructor of `empty_regex` is as follows: `Emp_Eps` specifies that the empty RE is equivalent to itself. For concatenation, choice and Kleene star, we can only say that they are equivalent to ϵ if all of its subterms are also equivalent to the empty RE.

Using the `empty_regex` predicate we can easily prove the following theorems. The first specifies that if `empty_regex e` holds then `e` accepts the empty string and the second says that if `empty_regex e` is provable then `e` is equivalent to the empty string RE.

```
Lemma empty_regex_sem : forall e, empty_regex e -> "" <<- e.
Theorem empty_regex_spec : forall e, empty_regex e -> e === #1.
```

The emptiness test function follows the same definition pattern as `null` using the `refine` tactic. We specify its type using `empty_regex` predicate and we omit its definition for brevity.

Having defined these two predicates, we can implement the function to convert problematic REs into non-problematic ones. The specification of when a RE is not problematic is given by the following inductive predicate.

```
Inductive unproblematic : regex -> Prop :=
| UEmpty : unproblematic #0
| UEps   : unproblematic #1
| UChr   : forall c, unproblematic ($ c)
| UCat   : forall e e', unproblematic e ->
                   unproblematic e' ->
                   unproblematic (e @ e')
| UChoice : forall e e', unproblematic e ->
                    unproblematic e' ->
                    unproblematic (e :+: e')
| UStar : forall e, ~ ("" <<- e) ->
                 unproblematic e ->
                 unproblematic (Star e).
```

Type `unproblematic` says that empty set, empty string and single characters REs are unproblematic. Concatenation and choice REs are unproblematic if both its subexpression are unproblematic. Finally, a Kleene star is unproblematic if its subexpression is unproblematic and does not accept the empty string. Finally, we specify the problematic RE conversion function with the following type:

```
Definition unprob
   : forall (e : regex), {e' | e === e' /\ unproblematic e'}.
```

Function `unprob` type says that from a input RE `e` it returns another RE `e'` which is unproblematic and equivalent to `e`. Again, we define `unprob` using `refine` tactic and its definition is just the Coq coding of $\texttt{f}_{\texttt{out}}$. As pointed by Medeiros et. al. [41], most of the work to produce a unproblematic RE is done by function $\texttt{f}_{\texttt{in}}$, which is applied when the inner RE of a Kleene star accepts the empty string and is not equivalent to the empty RE. Function `unprob_rec` implements $\texttt{f}_{\texttt{in}}$ function and we specify it with the following type:

```
Definition unprob_rec : forall e, "" <<- e -> ~ empty_regex e ->
    {e' | (e ^*) === (e' ^*) /\ ~ "" <<- e' /\ unproblematic e'}
```

`unprob_rec`'s type establishes that the return RE `e'` is unproblematic, does not accept the empty string and its Kleene star is equivalent to input REs Kleene star, i.e. `(e ^*) === (e' ^*)`.

Parse trees and bit-code representation In our formalization, we use the following inductive type to represent parse trees:

```
Inductive tree : Set :=
| TUnit  : tree | TChr   : ascii -> tree
| TCat   : tree -> tree -> tree
| TLeft  : tree -> tree | TRight : tree -> tree
| TNil   : tree | TCons  : tree -> tree -> tree.
```

Constructor `TUnit` denotes a parse tree for the empty string RE, `TChr` the tree for a single symbol RE and `TCat` the tree for the concatenation of two REs. `TLeft` and `TRight` denote trees for the choice operator. Constructors `TCons` and `TNil` can be used to form a list of trees for a Kleene star RE.

The parse tree typing judgment is coded as the following inductive predicate, in which each constructor has a correspondent rule in Figure 2.4.

```
Inductive is_tree_of : tree -> regex -> Prop :=
| ITUnit : TUnit :> #1
| ITChr  : forall c, (TChr c) :> ($ c)
| ITCat  : forall e t e' t',
    t :> e   ->
    t' :> e' ->
    (TCat t t') :> (e @ e')
| ITLeft : forall e t e',
   t :> e ->
   (TLeft t) :> (e :+: e')
| ITRight : forall e e' t',
   t' :> e' ->
   (TRight t') :> (e :+: e')
| ITNil : forall e, TNil :> (Star e)
| ITCons : forall e t ts,
  t :> e ->
  ts :> (Star e) ->
  (TCons t ts) :> (Star e)
where "t ':>' e" := (is_tree_of t e).
```

Function `flatten` has a direct encoding as a Coq recursive definition and we omit it for brevity. From `flatten` and tree typing relation definitions, theorems 2 and 3 are easily proved.

Bit coding of parse trees is represented by a list of bits, as follows:

```
Inductive bit : Set := O : bit | I : bit.
Definition code := list bit.
```

The typing relation for bit-coded parse trees (Figure 2.5) has an immediate definition as an inductively defined Coq relation.

```
Inductive is_code_of : code -> regex -> Prop :=
| ICEpsilon : [] :# #1
| ICChar    : forall c, [] :# ($ c)
| ICLeft    : forall bs e e'
  , bs :# e ->
    (O :: bs) :# (e :+: e')
| ICRight   : forall bs e e'
  , bs :# e' ->
    (I :: bs) :# (e :+: e')
| ICCat : forall bs bs' e e'
  , bs :# e ->
    bs' :# e' ->
    (bs ++ bs') :# (e @ e')
| ICNil : forall e, (I :: []) :# (e ^*)
| ICCons : forall e bs bss,
    bs :# e ->
    bss :# (e ^*) ->
    (O :: bs ++ bss) :# (e ^*)
where "bs ':#' e" := (is_code_of bs e).
```

As with `flatten`, function `code` has an immediate Coq definition. The next results about `code` are proved by a routine inductive proof.

```
Lemma encode_sound
  : forall bs e, bs :# e -> exists t, t :> e /\ encode t = bs.
Lemma encode_complete
  : forall t e, t :> e -> (encode t) :# e.
```

Unlike `code`, function `decode` has a more elaborated recursive definition, as shown in Section 2.4, since it recurses over the input RE while threading the remaining bits to be parsed into a tree. Since it has a more complicated definition, we use dependent types to combine its definition with its correctness proof. First, we define type `nocode_for` which denotes proofs that some bit list is not a valid bit-coded tree for some RE.

```
Inductive nocode_for : code -> regex -> Prop :=
| NCEmpty : forall bs, nocode_for bs #0
| NCChoicenil : forall e e', nocode_for [] (e :+: e')
| NCLBase : forall bs e e',
    nocode_for bs e ->
    nocode_for (O :: bs) (e :+: e')
| NCRBase : forall bs e e',
    nocode_for bs e' ->
    nocode_for (I :: bs) (e :+: e')
| NCStarnil : forall e, nocode_for [] (e ^*)
| NCStar : forall bs bs1 bs2 e,
      is_code_of bs1 e ->
      nocode_for bs2 (e ^*) ->
      bs = O :: bs1 ++ bs2  ->
      nocode_for bs (e ^*)
| NCStar1 : forall bs e,
    nocode_for bs e ->
    nocode_for (O :: bs) (e ^*).
(** some code omitted *)
```

Constructor `NCEmpty` specifies that there is no code for the empty set RE, ##0. For choice REs, we have several cases to cover. Constructor `NCChoicenil` specifies that the empty list is not a valid code for any choice RE. Constructor `NCLBase` (`NCRBase`) specifies that if a list isn't a valid code for a RE `e` (`e'`) it cannot be used to form a valid code for `e :+: e'`. In order to build a proof that some bit list isn't a valid code for a concatenation RE, we just need to prove that it is not a code for some of its subexpressions. Finally, for the Kleene star, we have some cases to cover: first, constructor `NCStarnil` shows that the empty list cannot be a code for any star RE. For non-empty bit-lists, it is just necessary to show that some part of the bit list isn't a code either for `e` or `e ^*`.

Using predicate `nocode_for` we can define a type for invalid bit-codes:

```
Definition invalid_code bs e :=
  nocode_for bs e \/ exists t b1 bs1, bs = (code t) ++ (b1 :: bs1).
```

which basically says that a bit list is an invalid code for a RE `e` when either we can construct a proof of `nocode_for` or we can parse a prefix of it into a valid tree but it leaves a non-empty bit list as a remaining suffix. Using this infrastructure, we can define the decode function with the following type:

```
Definition decode e bs :
   {t | bs = code t /\ is_tree_of t e} + {invalid_code bs e}.
```

Note that the previous type denotes the correctness property of a decode function: either it returns a valid tree for the input RE that can be converted into the input bit list or a proof that such bit list isn't a valid code for the input RE.

Formalizing the proposed semantics and its interpreter Our big-step semantics definition consists of the Coq representation of the judgment in Figure 3.3, which is presented below.

```
Inductive in_regex_p : string -> regex ->
                       string -> string -> Prop :=
| InEpsP
  : forall s, s <$- #1 ; "" ; s
| InChrP
  : forall a s,
    (String a s) <$- ($ a) ; (String a "") ; s
| InLeftP
  : forall s s' e e',
    (s ++ s') <$- e ; s ; s' ->
    (s ++ s') <$- (e :+: e') ; s ; s'
| InCatP : forall s s' s'' e e',
    (s ++ s' ++ s'') <$- e ; s ; (s' ++ s'') ->
    (s' ++ s'') <$- e' ; s' ; s'' ->
    (s ++ s' ++ s'') <$- (e @ e') ; (s ++ s') ; s''
| InStarRightP : forall s1 s2 s3 e,
    s1 <> "" ->
    (s1 ++ s2 ++ s3) <$- e ; s1 ; (s2 ++ s3) ->
    (s2 ++ s3) <$- (Star e) ; s2 ; s3 ->
    (s1 ++ s2 ++ s3) <$- (Star e) ; (s1 ++ s2) ; s3
where "s '<$-' e ';' s1 ';' s2" := (in_regex_p s e s1 s2).
(** some code omitted *)
```

In order to ease the task of writing types involving `in_regex_p`, we define the following notation `s <$- e ; s1 ; s2$` for `in_regex_p s e s1 s2`. The meaning of `in_regex_p` is the same as the rules of our semantics in Figure 3.3 and we omit redundant explanations for brevity.

The soundness and completeness theorems of the big-step semantics are stated below. Both are proved by induction on the complexity of the pair (e, s).

```
Theorem in_regex_p_complete :
   forall e s, s <<- e -> forall s', (s ++ s') <$- e ; s ; s'.
Theorem in_regex_p_sound :
   forall e s s1 s', s <$- e ; s1 ; s' ->
                     s = s1 ++ s' /\ s1 <<- e.
```

The completeness express that if an string `s` is in the language of RE `e`, i.e. `s <<- e`, then our semantics can parse the string `s ++ s'`, for any string `s'`. Soundness theorem says that whenever we have a derivation `s <$- e ; s1 ; s'`, then we have that the input string `s` should be equal to the concatenation of the parsed prefix (`s1`) and the remainder (`s'`), i.e. `s = s1 ++ s'`, and the parsed prefix should be in `e`'s language (`s1 <<- e`).

After a proper definition of our semantics, we developed a formalized interpreter for it. First, we need to define a type to store the intermediate results of the VM. We call this type `result` and its definition is shown below.

```
Record result : Set
  := Result {
          bitcode   : code
        ; consumed  : string
        ; remaining : string
    }.
```

Type `result` has an obvious meaning: it stores the computed bit-coded parse tree, the consumed prefix of the input string and its remaining suffix. Using type `result`, we can define the specification of our interpreter as:

```
Definition interp  : forall e s,
 {{r | exists e', unproblematic e' /\ e === e'  /\
   s = consumed r ++ remaining r /\
   (consumed r ++ remaining r) <$- e' ;consumed r ; remaining r /\
   (bitcode r) :# e'}}.
```

Function `interp` is defined as follows: first it converts the input RE into an equivalent unproblematic one and then proceed to parse the input string by well-founded recursion on the complexity of the pair (e, s). In its definition, we follow the same pattern used before: the computational content is specified using tactic `refine` to mark proof positions using holes that are filled later by tactics.

Extracting a certified implementation In order to obtain a certified Haskell implementation from our VM-based algorithm, we use Coq support for extraction, which has several pre-defined settings for using data-types and functions of Haskell's Prelude[4].

[4]Prelude is the name of the Haskell library automatically loaded in any Haskell module [29].

The extracted Haskell code for our VM interpreter has 259 lines. In order to use the algorithm, we build a grep-like command line tool, which is available at project's online repository [13].

3.2.4. Experimental Results

We use the formalized algorithm to build a Haskell tool for RE parsing and compare its performance against the library regex-applicative [9], a RE matching/parsing optimized library for Haskell. We chose it is because it allows us to build bit-coded parse trees using its applicative interface [39], enabling a more fair comparison with our algorithm. We ran our experiments on a machine with a Intel Core I7 1.7 GHz, 8GB RAM running Mac OS X 10.14.2; the results were collected and the average of several test runs were computed. In order to allow reproducibility, the on-line repository contains a Haskell program that automates the task of running the experiments to produce the graphs presented next.

Also, we would like to emphasize that the intent of these experiments is not to conclude that the proposed algorithm is more (less) efficient than the chosen library for RE parsing. Our main objective is to show that a fully verified algorithm can have a performance comparable to an optimized library to the same task.

The first experiment consists in parsing strings formed by a's by RE $(a + b + ab)^{\star}$ and the second with strings formed by ab's (examples taken from [50]). The results are presented in Figures 3.4 and 3.5.

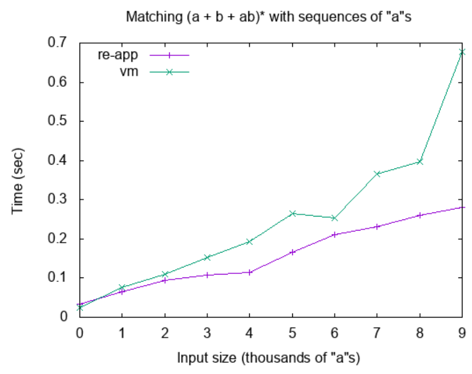


Figure 3.4.: Results of experiment 1.

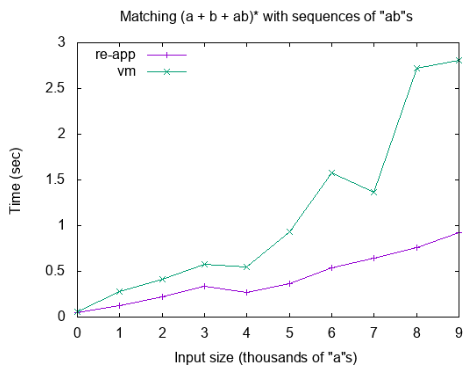


Figure 3.5.: Results of experiment 2.

When compared with regex-applicative, our tool exhibits a bad performance in this test (around 2 to 3 times slower than regex-applicative). The main reason for such behavior can be explained by the implementation details of regex-applicative, which internally compiles a RE to a NFA to parse the input string. Since the RE used in this test is short, converting it into a NFA does not influence in the library execution time.

Another experiment considered was to parse strings a^n by the RE $(a+\epsilon)^n a^n$, in which a^n denotes $n \geqslant 0$ copies of a. Such RE pose a chalenge to RE parsing algorithms since they need to simulate the traversal of 2^n paths, by backtracking, before finding a match [42]. The results of executing this experiments on increasing values of n is presented below.

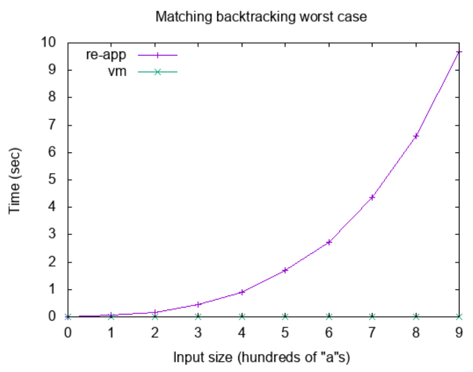


Figure 3.6.: Results of experiment 3.

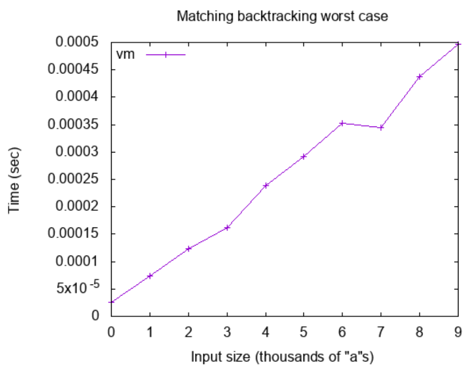


Figure 3.7.: Results of experiment 3 considering only the VM.

In this example, our approach has a much better performance than regex-applicative, which exhibits an exponential behavior (also known as catastrophic backtracking [30]). Such bad behavior on large REs can be explained by the NFA-based parsing algorithm used by regex-applicative library. Notice that our VM-based algorithm shows a linear performance on such problematic inputs, as presented in Figure 3.7.

The last experiment considered is to test how both approaches perform on random generated REs and random accepted strings for them. In order to perform such test, we use Haskell library QuickCheck [11]. The experiment consists in collecting the result of running both semantic on thousands of input pairs formed by a RE and strings. The average of such executions is presented in the Figure 3.8, which shows that both algorithms exhibit a linear behavior on random inputs.

A few words should be written about how we generate random inputs [5]. Generation of random RE is done by function `sizedRegex` with takes a depth limit to restrict the size of the generated RE. Whenever the input depth limit is less or equal to 1, we can only build a ϵ or a single character RE. The definition of `sizedRegex` uses QuickCheck function `frequency`, which receives a list of pairs formed by a weight and a random generator and produces, as result, a generator which uses such frequency distribution. In `sizedRegex` implementation we give a higher weight to generate characters and equal distributions to build concatenation, union or star.

```haskell
sizedRegex :: Int -> Gen Regex
sizedRegex n
  | n <= 1 = frequency [ (10, return Eps), (90, Chr <$> genChar) ]
  | otherwise = frequency [ (10, return Epsilon), (30, Chr <$> genChar)
         , (20, Cat <$> sizedRegex n2 <*> sizedRegex n2)
```

[5]We assume that the reader has some acquaintance with Haskell programming language and its library for property based testing, QuickCheck. Good introductions to Haskell are available elsewhere [34].

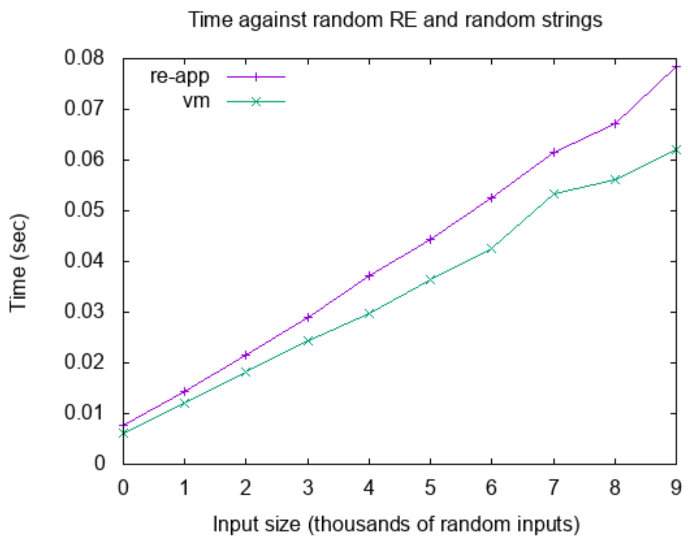


Figure 3.8.: Results of experiment 4.

```
        , (20, Choice <$> sizedRegex n2 <*> sizedRegex n2)
        , (20, Star   <$> sizedRegex n2)]
        where n2 = div n 2
```

Given an RE e, we can generate a random string s such that $s \in [\![e]\!]$ using the next definition. We generate strings by choosing randomly between branches of a union or by repeating n times a string s which is accepted by e, whenever we have $e^\star$ (function randomMatches).

```
randomMatch :: Regex -> Gen String
randomMatch Eps = return ""
randomMatch (Chr c) = return [c]
randomMatch (Cat e e') = liftM2 (++) (randomMatch e)
                                     (randomMatch e')
randomMatch (Choice e e') = oneof [ randomMatch e, randomMatch e' ]
randomMatch (Star e) = do
      n <- choose (0,3) :: Gen Int
      randomMatches n e

randomMatches :: Int -> Regex -> Gen String
randomMatches m e'
  | m <= 0 = return []
  | otherwise = liftM2 (++) (randomMatch e')
                            (randomMatches (m - 1) e')
```

4. Related Works

A new technique for constructing a finite deterministic automaton from a RE was presented by Asperti et al. in [3]. It's based on the idea of marking a suitable set of positions inside the RE, intuitively representing the possible points reached after the processing of an initial prefix of the input string. In other words, the points mark positions inside the RE which have been reached after reading some prefix of the input string, or better positions where the processing of remaining string has to be started. Each pointed expression for a RE e represents a state of the deterministic automaton associated with e; since there is obviously only a finite number of possible labellings, the number of states of the automaton is finite. The authors argued that Pointed REs join the elegance and the symbolic appealingness of Brzozowski's derivatives with the effectiveness of McNaughton and Yamada's labelling technique, essentially combining the best of both approaches, allowing a direct, intuitive and easily verifiable construction of the deterministic automaton for e. The authors said that pointed expressions can provide a more compact description for RLs than traditional REs. However, the authors do not discuss the usage of pointed REs for parsing or matching.

Brüggemann-Klein [8] showed that the Glushkov automaton can be constructed in a quadratic time in the size of a RE, and that this is worst-case optimal and output sensitive. For deterministic REs, her algorithm has even linear run time. This improves on the cubic methods suggested in the literature. Although her paper was focused on time complexity, the author stated - based on one of her references - that strong unambiguity of REs can be reduced in linear time to unambiguity of ϵ-NFA's via Thompson's construction, which is the one we based to do this work.

The concept of prioritized transducers to formalize capturing groups in RE matching was introduced by Berglund and Merwe [5]. Their main goal was to provide an automaton-based theoretical foundation for the basic functionality of modern RE matchers (with focus on the Java RE standard library). Many RE matching libraries perform matching as a way of parsing by using capturing groups, and thus output what subexpression matched which sub-string. Their approach permits an analysis of matching semantic of a subset of REs supported in Java. According to the authors, converting REs to what they called as prioritized transducers is a natural generalization of the Thompson construction for REs to NFA.

A method of obtaining a ϵ-free automaton from RE is presented by García et al. [22]. In their proposal, the number of states of the obtained automaton is bounded above by the size of both the partial derivatives (Antimirov) and the follow automata (Illie an Yu [28]). Their algorithm also runs with the same time complexity of those methods. Although they mentioned Thompson's automaton as one of the first methods to do the task of representing REs as automata, their work did not present any formal proof about the correctness of their proposed algorithm. Their main concern seemed to be the efficiency of their algorithm, not its correctness.

Berry and Sethi [6] presented a study about two well-known algorithms for constructing a finite automaton from a RE. Their main idea is to allow an elegant algorithm to be refined into an efficient one. The elegant algorithm is based on 'derivatives' of REs; the efficient one is based on 'marking of' REs. They showed proofs that it is possible to move from the derivative approach to the marking one without losing the benefits of both approaches. However, intersection and complement (which are additional operators for REs) cannot be handled because the marking and unmarking processes do not preserve the languages generated by REs with these operators.

A formal constructive theory of RLs was presented by Doczkal et al. in [15]. They formalized some fundamental results about RLs. For their formalization, they used the SSReflect extension to Coq, which features an extensive library with support for reasoning about finite structures such as finite types and finite graphs. They established all of their results in about 1400 lines of Coq, half of which are specifications. Most of their formalization deals with translations between different representations of RLs, including REs, DFAs, minimal DFAs and NFAs. They formalized all these (and other) representations and constructed computable conversions between them. Besides other interesting aspects of their work, they proved the decidability of language equivalence for all representations. Unlike our work, Doczkal et al.'s only concerns about formalizing classical results of RL theory in Coq, without using the formalized automata in practical applications, like matching or parsing.

Groz and Maneth [24] approached the efficiency of testing and matching of deterministic REs. They presented a linear time algorithm for testing whether a RE is deterministic and an efficient algorithm for matching words against deterministic REs. It was shown that an input word of length n can be matched against a deterministic RE of length m in time $O(m + n \log\log m)$. If the deterministic RE has bounded depth of alternating union and concatenation operators, then matching can be performed in time $O(m + n)$. According to the authors, these results extend to REs containing numerical occurrence indicators. The authors presented the concept of deterministic REs and the differences between weak and strong determinism. Their paper contains some proofs, many of them related to algorithmic running time. However, their approach was focused on performance over deterministic REs, leaving aside the non-deterministic ones. We intend to investigate time complexity of algorithm in future works.

Radanne and Thiemann [45] pointed that some of the algorithms for RE matching are rather intricate and the natural question that arises is how to test these algorithms. It is not too hard to come up with generators for strings that match a given RE, but on the other hand, the algorithms should reject strings that do not match that RE. So it is equally important to come up with strings that do not match. In other words, a satisfactory solution for testing such matchers would require generating positive as well as negative examples for some language. Thus, the authors presented an algorithm to generate the language of a generalized RE with union, intersection and complement operators. Using this technique, they could generate both positive and negative instances of a RE. They provided two implementations: one in Haskell, which explores different algorithmic improvements, and one in OCaml, which evaluates choices in data structures. Their algorithm lacks of correctness proofs.

Ilie and Yu [28] presented two algorithms for constructing non-deterministic finite automata (NFA) from REs. The first one constructs NFAs with ϵ-transitions (ϵ-NFA), which are smaller than all other ϵ-NFAs obtained by similar constructions. The second one constructs NFAs by removing ϵ-elimination in ϵ-NFAs and builds a quotient of the well-known position automaton with respect to the equivalence given by the follow relation, named by the authors as *follow automaton*, which uses optimally the information from positions of a RE. The authors compared follow automaton with the best existing constructions in their time (position, partial derivative, and common follow sets automata) and concluded that their follow automaton has interesting properties: it is always a quotient of the position automaton, it is very easy to compute and it is at least as small as all the other similarly constructed automata in most cases. Among the several problems pointed by the authors that should be investigated further, it should be done a more rigorous comparison between the follow automaton and common follow sets or partial derivative automaton. According to the authors, "probably the only way to decide which one is better is by testing all of them in real-life applications".

Spivey approached the theme of parser combinators in [49]. The main idea is that a parser of phrases of a type α is a function that takes an input string and produces results $(x, rest)$ in which x is a value of type α and $rest$ is the remainder of the input after the phrase with value x has been consumed. The results are often arranged into a list, because this allows a parser to signal failure with the empty list of results, an unambiguous success with one result, or multiple possibilities with a longer 'list of successes' [1]. Producing a list of results naturally leads to backtracking parsers that can be exponentially slow, so it is preferable when possible to replace the type *List* by a different parser type, known in Haskell as type *Maybe*. The use of this parser type reduces the amount of fruitless searching and permits the record of choices made in recognizing a phrase to be discarded as soon as one of the choices succeeds. In an unambiguous grammar, an input string will either fail to be in the language or will have exactly one derivation tree. The author says that a parser *works correctly* if it has type *List* and it returns $[\,]$ and $[(x, ``")]$ or if it has type *Maybe* and returns $Nothing$ and $Just(x, ``")$ in both cases. According to the author, both *List* and *Maybe* types work correctly for any grammar that has no left recursion. On the other hand, grammars that are $LL(1)$ can be parsed with no backtracking at all and both types of parsers work correctly. The result reported in Spivey's paper is that it is not decidable whether a $Maybe$-based parser will continue to work correctly for cases in which the grammars 'are not quite $LL(1)$'.

The main goal of Medeiros et al. [37] is to present a new formalization of REs via transformation to PEGs and to show that their formalization accommodates some of regex [2] extensions. They present formalizations of both REs and PEGs in the framework of natural semantics and use these to show the similarities and differences between REs

[1] We will call this approach as type *List*

[2] Regexes add several ad-hoc extensions to REs. They may look like REs, but can have syntactical and semantical extensions that are difficult - or impossible - to express through pure REs.

and PEGs. Then, they define a transformation that converts a RE to a PEG and prove its correctness. Finally, they show how this transformation can be adapted to accommodate some regex extensions. One of many interesting points of their work is that they also show how to obtain a well-formed [3] RE that recognizes the same language as non-well-formed REs (details in Subsection 3.2.1).

Ierusalimschy [27] proposed the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. He argued that pure REs have proven to be a too weak formalism for that task: many interesting patterns either are difficult to describe or cannot be described by REs. He also said that the inherent non-determinism of REs does not fit the need to capture specific parts of a match. Following this proposal, he presented LPEG, a pattern-matching tool based on PEGs for the Lua scripting language. He argued that LPEG unifies the ease of use of pattern-matching tools with the full expressive power of PEGs. He also presented a parsing machine (PM) that allows an implementation of PEGs for pattern matching. The author presented no proofs of the PM's correctness. Besides, there is no guarantee that his LPEG implementation follows his specification.

However, Medeiros and Ierusalimschy [40] presented a new approach for implementing PEGs, based on a virtual parsing machine (VM). Each PEG has a corresponding program that is executed by the parsing machine, and new programs are dynamically created and composed. They gave an operational semantics of PEGs used for pattern matching, then described their parsing machine and its semantics. They showed how to transform PEGs to parsing machine programs, and gave a correctness proof of their compiler transformation. This work is more similar to ours, once that we also intend to develop a VM for parsing and prove its correctness. However, the proofs presented by those authors were not verified by a proof assistant. They said that the execution model of their machine cannot handle infinite loops. Furthermore, the "star" operator for PEGs has a greedy semantics which differs from the conventional RE semantics for this operator.

Rathnayake and Thielecke [46] formalized a VM implementation for RE matching using operational semantics. Specifically, they derived a series of abstract machines, moving from the abstract definition of matching to realistic machines. First, a continuation is added to the operational semantics to describe what remains to be matched after the current expression. Next, they represented the expression as a data structure using pointers, which enables redundant searches to be eliminated via testing for pointer equality. Although their work has some similarities with ours (a VM-based RE parsing algorithm), they did not present any evidence or proofs that their VM is correct.

Fischer, Huch and Wilke [19] developed a Haskell program for matching REs. The program is purely functional and it is overloaded over arbitrary semirings, which solves the matching problem and supports other applications like computing leftmost longest matchings or the number of matchings. Their program can also be used for parsing every context-free language by taking advantage of laziness. Their developed program is based on an old technique to turn REs into finite automata, which makes it efficient

[3]A RE e that has a subexpression e_i^* where e_i can match the empty string is not well-formed.

compared to other similar approaches. One advantage of their implementation over our proposal is that their approach works with context-free languages, not only with REs purely. However, they did not present any correctness proof of their Haskell code.

Cox [12] said that viewing RE matching as executing a special machine makes it possible to add new features just by the inclusion of new machine instructions. He presented two different ways to implement a VM that executes a RE that has been compiled into byte-codes: a recursive and a non-recursive backtracking implementation, both in C programming language. Cox's work on VM-based RE parsing is poorly specified: both the VM semantics and the RE compilation process are described only informally and no correctness guarantees are even mentioned.

Frisch and Cardelli [21] studied the theoretical problem of matching a flat sequence against a type (RE): the result of the process is a structured value of a given type. Their contributions were in noticing that: (1) A disambiguated result of parsing can be presented as a data structure that does not contain ambiguities. (2) There are problematic cases in parsing values of star types that need to be disambiguated. (3) The disambiguation strategy used in XDuce and CDuce (two XML-oriented functional languages) pattern matching can be characterized mathematically by what they call greedy RE matching. (4) There is a linear time algorithm for the greedy matching. Their approach is different since they want to axiomatize abstractly the disambiguation policy, without providing an explicit matching algorithm. They identified three notions of problematic words, REs, and values (which represent the ways to match words), related these three notions, and proposed matching algorithms to deal with the problematic case.

Ribeiro and Du Bois [47] described the formalization of a RE parsing algorithm that produces a bit representation of its parse tree in the dependently typed language Agda. The algorithm computes bit-codes using Brzozowski derivatives and they proved that the produced codes are equivalent to parse trees ensuring soundness and completeness with respect to an inductive RE semantics. They included the certified algorithm in a tool developed by themselves, named verigrep, for RE-based search in the style of GNU Grep. While the authors provided formal proofs, their tool showed a poor performance when compared with other approaches to RE parsing.

Nielsen and Henglein [42] showed how to generate a compact bit-coded representation of a parse tree for a given RE efficiently, without explicitly constructing the parse tree first, by simplifying the DFA-based parsing algorithm of Dubé and Feeley [16] to emit a bit representation without explicitly materializing the parse tree itself. They also showed that Frisch and Cardelli's greedy RE parsing algorithm [21] can be straightforwardly modified to produce bit codings directly. They implemented both solutions as well as a backtracking parser and performed benchmark experiments to measure their performance. They argued that bit codings are interesting in their own right since they are typically not only smaller than the parse tree, but also smaller than the string being parsed and can be combined with other techniques for improved text compression. As others related works, the authors did not present a formal verification of their implementations.

A recent application of REs was presented by Radanne [44]. In many cases, the goal of a RE is not only to match a given text, but also to extract information from it. With that in mind, the author presented a technique to provide type-safe extraction based on the typed interpretation of REs. That technique relies on two-layer REs in which the upper layer allows to compose and transform data in a well-typed way, while the lower one is composed by untyped REs that can leverage features from a preexisting RE matching engine. Results showed that this technique is faster than other two libraries that perform the same task, despite its lack of efficiency when compared with some full RE parsing algorithms. No formalization was provided in that work.

Sulzmann et al. [50] proposed an algorithm for POSIX RE parsing with uses RE derivatives to construct parse trees incrementally to solve both matching and submatching for REs. In order to improve the efficiency of the proposed algorithm, Sulzmann et al. used a bit encoded representation of RE parse trees. Ausaf et. al. [4] present a Isabelle/HOL formalization of Sulzmann et. al POSIX parsing algorithm. They gave their inductive definition of what a POSIX value is and showed that such a value is unique for a given RE and a string being matched. We intend, as future work, to use a similar inductive definition to characterize the disambiguation strategy followed by our VM semantics.

5. Conclusion

In this work, we developed a semantic characterization of a VM based algorithm for RE parsing. For it, we proposed two operational semantics. The small-step operational semantics contains more details about how a RE should be evaluated and produces bit-codes as parsing evidence. A prototype for it was developed in Haskell and then we submitted it to a property-based testing with QuickCheck. The results gave us a strong evidences of the semantics' correctness, although it could not deal with problematic REs. Later, we presented an evolution of the small-step semantics - a big-step operational semantics for the proposed algorithm, which also produces bit-codes. In order to avoid the well-known problems with problematic REs, this time we used an algorithm that converts a problematic RE into an equivalent non-problematic one. All theoretical results reported for the big step semantics are integrally verified using Coq proof assistant. From our formalization, we extract a Haskell implementation of our algorithm and used it to build a tool for RE parsing, which has performance comparable to an optimized Haskell library for RE parsing. The complete development is available at [13].

As future work, we intend to extend our semantics with some real-world regex features like capture groups and quantifiers, while keeping an easy to follow formalization and an efficient algorithmic interpreter for it. Another line of research we intend to pursue is to formalize that the proposed semantics follows a disambiguation strategy and to investigate the time complexity of our algorithm.

Bibliography

[1] Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings.*, pages 16–30, 2003.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. Regular expressions, au point. *CoRR*, abs/1010.2604, 2010.

[4] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. Posix lexing with derivatives of regular expressions (proof pearl). In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving*, pages 69–86, Cham, 2016. Springer International Publishing.

[5] Martin Berglund and Brink Van Der Merwe. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science*, 1:1–14, 2016.

[6] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(C):117–126, 1986.

[7] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[8] Anne Brüggemann-Klein. Regular expressions into finite automata. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 583 LNCS:87–98, 1992.

[9] Roman Cheplyaka. regex-applicative: Regex based parsing with applicative interface — on-line repository. http://hackage.haskell.org/package/regex-applicative, 2018.

[10] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013.

[11] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International*

Conference on Functional Programming, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[12] Russ Cox. Regular Expression Matching: the Virtual Machine Approach. 2009.

[13] Thales Delfino and Rodrigo Ribeiro. Towards certified virtual machine-based regular expression parsing — on-line repository. https://github.com/thalesad/regexvm, 2018.

[14] Thales Antônio Delfino and Rodrigo Ribeiro. Towards certified virtual machine-based regular expression parsing. In *Proceedings of the XXII Brazilian Symposium on Programming Languages*, SBLP '18, pages 67–74, New York, NY, USA, 2018. ACM.

[15] Christian Doczkal, Jan Oliver Kaiser, and Gert Smolka. A constructive theory of regular languages in Coq. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8307 LNCS:82–97, 2013.

[16] Danny Dubé and Marc Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, Sep 2000.

[17] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[18] Denis Firsov and Tarmo Uustalu. Certified parsing of regular languages. In *Proceedings of the Third International Conference on Certified Programs and Proofs - Volume 8307*, pages 98–113, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[19] Sebastian Fischer, Frank Huch, and Thomas Wilke. A play on regular expressions. *ACM SIGPLAN Notices*, 45(9):357, 2010.

[20] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[21] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, pages 618–629, 2004.

[22] Pedro García, Damián López, José Ruiz, and Gloria I. Álvarez. From regular expressions to smaller NFAs. *Theoretical Computer Science*, 412(41):5802–5807, 2011.

[23] Andy Gill and Colin Runciman. Haskell program coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, pages 1–12, New York, NY, USA, 2007. ACM.

[24] B. Groz and S. Maneth. Efficient testing and matching of deterministic regular expressions. *Journal of Computer and System Sciences*, 89:372–399, 2017.

[25] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[26] Graham Hutton. Fold and Unfold for Program Semantics. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*, Baltimore, Maryland, September 1998.

[27] Roberto Ierusalimschy. A Text Pattern-Matching Tool based on Parsing Expression Grammars. *Software - Practice and Experience*, 2009.

[28] Lucian Ilie and Sheng Yu. Follow automata. *Information and Computation*, 186(1):140–162, 2003.

[29] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[30] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, pages 135–148, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[31] Donald E. Knuth. Top-down syntax analysis. *Acta Inf.*, 1(2):79–110, June 1971.

[32] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbol. Comput.*, 20(3):199–207, September 2007.

[33] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[34] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.

[35] Raul Lopes, Rodrigo Ribeiro, and Carlos Camarão. Certified derivative-based parsing of regular expressions. In *Programming Languages — Lecture Notes in Computer Science 9889*, pages 95–109. Springer, 2016.

[36] Raul Felipe Pimenta Lopes. Certified derivative-based parsing of regular expressions. Master's thesis, Federal University of Ouro Preto, Ouro Preto, MG, 2018.

[37] Fabio Mascarenhas and Roberto Ierusalimschy. From Regular Expressions to Parsing Expression Grammars. *Brazilian Symposium on Programming Languages*, 2011.

[38] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 287–295, 2008.

[39] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[40] Sérgio Medeiros and Roberto Ierusalimschy. A parsing machine for PEGs. *Proceedings of the 2008 symposium on Dynamic languages - DLS '08*, pages 1–12, 2008.

[41] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. From regexes to parsing expression grammars. *Sci. Comput. Program.*, 93:3–18, November 2014.

[42] Lasse Nielsen and Fritz Henglein. Bit-coded regular expression parsing. In Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide, editors, *Language and Automata Theory and Applications*, pages 402–413, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[43] Benjamin Pierce. *Types and Programming Languages*, volume 35. 2000.

[44] Gabriel Radanne. Typed parsing and unparsing for untyped regular expression engines. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2019, pages 35–46, New York, NY, USA, 2019. ACM.

[45] Gabriel Radanne and Peter Thiemann. Regenerate: A Language Generator for Extended Regular Expressions. working paper or preprint, May 2018.

[46] Asiri Rathnayake and Hayo Thielecke. Regular Expression Matching and Operational Semantics. *Electronic Proceedings in Theoretical Computer Science*, 62(Sos):31–45, 2011.

[47] Rodrigo Ribeiro and André Du Bois. Certified Bit-Coded Regular Expression Parsing. *Proceedings of the 21st Brazilian Symposium on Programming Languages - SBLP 2017*, pages 1–8, 2017.

[48] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA, 2006.

[49] Michael Spivey. When Maybe is not good enough. *Journal of Functional Programming*, 22(06):747–756, 2012.

[50] Martin Sulzmann and Kenny Zhuo Ming Lu. Posix regular expression parsing with derivatives. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 203–220, Cham, 2014. Springer International Publishing.

[51] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

A. Correctness of the accept Function

Fisher et. al. [19] presents a simple and elegant function for parsing a string using a RE. It relies on two auxiliary functions that break an input string into its parts. The first is function split which decompose the input string in a prefix and a suffix.

```
split::[a] → [([a], [a])]
split [] = [([], [])]
split (c : cs) = ([], c : cs) : [(c : s1, s2) | (s1, s2) ← split cs]
```

Function split has the following correctness property.

Lemma 3. *Let* xs *be an arbitrary list. For all* ys, zs *such that* (ys, zs) ∈ split xs*, we have that* xs ≡ ys ++ zs.

Proof. By induction on the structure of xs. □

Function parts decomposes a string into a list of its parts. Such property is expressed by the following lemma.

Lemma 4. *Let* xs *be an arbitrary list. For all* yss *such that* yss ∈ parts xs*, we have that* concat yss ≡ xs.

Proof. By induction on the structure of xs. □

Finally, function accept is defined by recursion on the input RE using functions parts and split in the Kleene star and concatenation cases. The correctness of accept states that it returns true only when the input string is in input RE's language, as stated in the next theorem.

Theorem 14. *For all* s *and* e, accept e s ≡ True *if, and only if,* $s \in [\![e]\!]$.

Proof.

(→) : By induction on the structure of e using lemmas about parts and split.

(←) : By induction on the derivation of $s \in [\![e]\!]$.

□