Towards certified virtual machine-based regular expression parsing

Thales Antônio Delfino
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais, Brazil

Rodrigo Ribeiro
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais, Brazil

Abstract

Regular expressions (REs) are pervasive in computing. We use REs in text editors, string search tools (like GNU-Grep) and lexical analysers generators. Most of these tools rely on converting regular expressions to its corresponding finite state machine or use REs derivatives for directly parse an input string. In this work, we investigate the suitability of another approach: instead of using derivatives or generate a finite state machine for a given RE, we developed a virtual machine (VM) for parsing regular languages, in such a way that a RE is merely a program executed by the VM over the input string. We provided a prototype implementation in Haskell, tested it using QuickCheck and provided proof sketches of its correctness with respect to RE standard inductive semantics.

CCS Concepts **•Theory of computation → Regular languages; Operational semantics;**

Keywords Regular Expressions, Parsing, Virtual Machines, Operational semantics

ACM Reference format:
Thales Antônio Delfino and Rodrigo Ribeiro. 2018. Towards certified virtual machine-based regular expression parsing. In *Proceedings of XXII BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES, São Carlos, September 17–21, 2018 (SBLP2018),* 15 pages.
DOI: 10.475/123_4

1 Introduction

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 analyser generators and XML schema languages [14]. Since RE parsing is pervasive in computing, its correctness is crucial and is the subject of study of several recent research works (e.g [3, 11, 21, 28]).

Approaches for RE parsing can use representations of finite state machines (e.g. [11]), derivatives (e.g. [21, 22, 28]) or the so-called pointed RE's or its variants [3, 12]. Another approach for parsing is based on the so-called parsing machines, which dates back to 70's with Knuth's work on top-down syntax analysis for context-free languages [18]. Recently, some works have tried to revive the use of such machines for parsing: Cox [7] 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 [17] defines a VM whose instruction set can be used to compile Parser Expressions Grammars (PEGs) [13]. 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 is even mentioned. Second, it does not provide an evidence for matching, which could be used to characterize a disambiguation strategy, like Greedy [14] and POSIX [30]. To the best of our knowledge, no previous work has formally defined a VM for RE parsing that produces evidence (parse trees) for successul matches. The objective of this work is to give a first step in filling this gap. More specifically, we are interested in formally specify, implement and test the correctness of a VM based small-step semantics for RE parsing which produces bit-codes as a memory efficient representation of parse-trees. As pointed by [26], 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 leave the task of proving that our VM follows a specific disambiguation strategy to future work.

Our contributions are:

- We present a small-step semantics for RE inspired by Thompson's NFA[1] construction [31]. 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[2] that our semantics is sound and complete with respect to RE inductive semantics.
- We describe a prototype implementation of our semantics in Haskell and use QuickCheck [6] to test our semantics against a simple implementation of RE parsing, presented in [12], which we prove correct in the Appendix A. Our test cases cover both accepted and rejected strings for randomly generated REs.

[1]Non-deterministic finite automata.
[2]By "informal proofs" we mean proofs that are not mechanized in a proof-assistant. Due to space reasons, proofs of the relevant theorems are omitted from this version. Detailed proofs can be found in the accompanying technical report avaliable on-line [8].

- We show how our proposed semantics can produce bit codes that denote parse trees [26] and test that such generated codes correspond to valid parsing evidence using QuickCheck.

We are aware that using automated testing is not sufficient to ensure correctness, but it can expose bugs before using more formal approaches, like formalizing our algorithm in a proof assistant. Such semantic prototyping step is crucial since it can avoid proof attempts that are doomed to fail due to incorrect definitions. The project's on-line repository [8] contains the partial Coq formalization of our semantics. Currently, we have formalized the semantics and its interpreter function. The Coq proof that the proposed small-step semantics is equivalent to the usual inductive RE semantics is under development.

The rest of this paper is organized as follows. Section 2 presents some background concepts on RE and data type derivatives that will be used in our semantics. Our operational semantics for RE parsing and its theoretical properties are described in Section 3. Our prototype implementation and the QuickCheck test suit used to validate it are presented in Section 4. Section 5 discuss related work and Section 6 concludes.

We assume that the reader knows the Haskell programming language, specially the list monad and how it can be used to model non-determinism. Good introductions to Haskell are available elsewhere [19]. All source code produced, including the literate Haskell source of this article (which can be preprocessed using lhs2TEX [20]), instructions on how to build it and reproduce the developed test suit are avaliable on-line [8].

2 Background

2.1 Regular expressions: syntax and semantics

REs are defined with respect to a given alphabet. Formally, the following context-free grammar defines RE syntax:

$$e ::= \emptyset \mid \epsilon \mid a \mid e\,e \mid e + e \mid e^\star$$

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 Haskell implementation, we represent alphabet symbols using type Char.

```
data Regex = ∅ | ϵ | Chr Char | Regex • Regex
   | Regex + Regex | Star Regex
```

Constructors $\emptyset$ and ϵ denote respectively the empty set ($\emptyset$) and the empty string (ϵ) REs. Alphabet symbols are constructed by using the Chr constructor. Bigger REs are built using concatenation (•), union (+) and Kleene star (Star).

Following common practice [21, 27, 28], we adopt an inductive characterization of RE membership semantics. We let judgment $s \in [\![e]\!]$ denote that string s is in the language denoted by RE e.

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 RL for Chr 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: $e^\star = \epsilon + e\,e^\star$.

We say that a RE e is *problematic* if $e = e'^\star$ and $\epsilon \in [\![e']\!]$ [14]. In this work, we limit our attention to non-problematic RE's. Our

$$\frac{}{\epsilon \in [\![\epsilon]\!]}\{Eps\} \qquad \frac{a \in \Sigma}{a \in [\![a]\!]}\{Chr\}$$

$$\frac{s \in [\![e]\!]}{s \in [\![e + e']\!]}\{Left\} \qquad \frac{s' \in [\![e']\!]}{s' \in [\![e + e']\!]}\{Right\}$$

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

$$\frac{s \in [\![e]\!] \quad s' \in [\![e']\!]}{ss' \in [\![ee']\!]}\{Cat\}$$

Figure 1. RE inductive semantics.

results can be extended to problematic REs without providing any new insight [14, 26].

2.2 RE parsing and bit-coded parse trees

RE parsing. One way to represent parsing evidence is to build a tree that denotes a RE membership proof. Following [14, 26], we let parse trees be terms whose type is underlying RE.

```
data Tree = () | Chr Char | Tree • Tree | InL Tree
   | InR Tree | List [Tree]
```

Constructor () denotes a tree for RE ϵ and Chr is a tree for a single character RE. Trees for concatenations are pairs, constructors InL and InR denotes trees for the left and right component of a choice operator. Finally, a tree for RE $e^\star$ is a list of trees for RE e. This informal relation is specified by the following inductive relation between parse trees and RE. We let $\vdash t : e$ denote that t is a parse tree for RE e.

$$\frac{}{\vdash () : \epsilon} \qquad \frac{}{\vdash \mathsf{Chr}\ \mathsf{a} : a} \qquad \frac{\vdash \mathsf{t} : e}{\vdash \mathsf{InL}\ \mathsf{t} : e + e'}$$

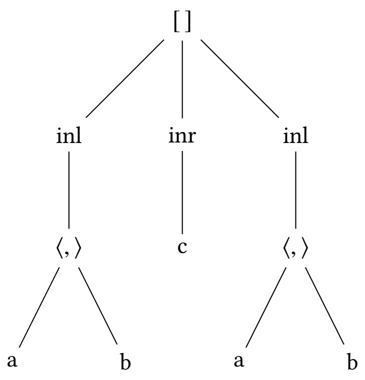
$$\frac{\vdash \mathsf{t}' : e'}{\vdash \mathsf{InR}\ \mathsf{t}' : e + e'} \qquad \frac{\vdash \mathsf{t} : e \quad \vdash \mathsf{t}' : e'}{\vdash \mathsf{t} \bullet \mathsf{t}' : ee'} \qquad \frac{\forall \mathsf{t}.\mathsf{t} \in \mathsf{ts} \rightarrow \vdash \mathsf{t} : e}{\vdash \mathsf{List}\ \mathsf{ts} : e^\star}$$

Figure 2. Parse tree typing relation.

The relation between RE semantics and its parse trees are formalized using the function flat, which builds the string stored in a given parse tree. The Haskell implementation of flat is immediate.

```
flat :: Tree → String
flat () = ""
flat (Chr c) = [c]
flat (t • t′) = flat t ++ flat t′
flat (InL t) = flat t
flat (InR t) = flat t
flat (List ts) = concatMap flat ts
```

Example 1. 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 theorem, which relates parse tress and RE semantics, can be proved by an easy induction on the RE semantics derivation.

Theorem 1. *For all s and e, if $s \in [\![e]\!]$ then exists a tree* t *such that* flat t $= s$ *and* $\vdash$ t $: e$.

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

1. Case rule (Eps): Then, $s = \epsilon$. Let $t = ()$ and the conclusion follows by the definition of flat and rule $T1$.
2. Case rule (Chr): Then, $s = a$, $a \in \Sigma$. Let $t =$ Chr a and the conclusion follows by the definition of flat and rule $T2$.
3. Case rule ($Left$): Then, $e = e_1 + e_2$ and $s \in [\![e_1]\!]$. By the induction hypothesis, we have a tree t_l such that flat tl = s and $\vdash t_l : e_1$. Let t = InL tl. Conclusion follows from rule $T3$ and the definition of flat.
4. Case rule ($Right$): Then, $e = e_1 + e_2$ and $s \in [\![e_2]\!]$. By the induction hypothesis, we have a tree t_r such that flat tr = s and $\vdash t_r : e_2$. Let t = InR tr. Conclusion follows from rule $T4$ and the definition of flat.
5. Case rule (Cat): Then, $e = e_1\, e_2$, $s = s_1\, s_2$, $s_1 \in [\![e_1]\!]$ and $s_2 \in [\![e_2]\!]$. By the induction hypothesis we have trees t_1 and t_2 such that flat t1 $=s_1$, $\vdash t_1 : e_1$, flat t2 $=s_2$ and $\vdash t_2 : e_2$. Let t = t1 • t2. Conclusion follows by rule $T5$ and the definition of flat.
6. Case rule ($StarBase$): Then, $e = e_1^\star$, $s = \epsilon$. Let t = List [ ]. Conclusion follows by rule $T6$ and the definition of flat.
7. Case rule ($StarRec$): Then, $e = e_1^\star$, $s = s_1\, s_2$, $s_1 \in [\![e_2]\!]$ and $s_2 \in [\![e_2]\!]$. By induction hypothesis, we have trees t_1 and t_2 such that flat t1 = s1, $\vdash t_1 : e_1$, flat t_2 = s2, $\vdash t2 : e_1^\star$ and t2 = List ts, for some list ts. Let t = List (t1:ts). Conclusion follows from the definition of flat and rule $T6$.

□

Bit-coded parse trees. Nielsen et. al. [26] 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. [28] showed an inductively defined relation between valid bit-codes and RE, accordingly to the encoding proposed by [26]. We let the judgement $bs \rhd e$ denote that the sequence of bits bs corresponds to a parse-tree for RE e.

The empty string and single character RE are both represented by empty bit lists. Codes for RE ee' 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 $e^\star$.

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

$$\frac{\text{bs} \rhd e'}{1_b : \text{bs} \rhd e + e'} \qquad \frac{\text{bs} \rhd e \quad \text{bs}' \rhd e'}{\text{bs} +\!\!+ \text{bs}' \rhd ee'} \qquad \frac{}{[1_b] \rhd e^\star}$$

$$\frac{\text{bs} \rhd e \quad \text{bss} \rhd e^\star}{0_b : \text{bs} +\!\!+ \text{bss} \rhd e^\star}$$

Figure 3. Typing relation for bit-codes.

The relation between a bit-code and its underlying parse tree can be defined using functions code and decode. Type Code used in code and decode definition is just a synonym for [Bit]. Function code has an immediate definition by recursion on the structure of parse tree.

```
code :: Tree → Regex → Code
code (InL t) (e + _) = 0b : code t e
code (InR t′) (_ + e′) = 1b : code t′ e′
code (List ts) (Star e) = codeList ts e
code (t • t′) (e • e′) = code t e ++ code t′ e′
code _ _ = [ ]
codeList :: [Tree] → Regex → Code
codeList ts e = foldr (λt ac → 0b : code t e ++ ac) [1b] ts
```

To define function decode, we need to keep track of the remaining bits to be processed to finish tree construction. This task is done by an auxiliar definition, dec.

```
dec :: Regex → Code → Maybe (Tree, Code)
dec ϵ bs = return ((), bs)
dec (Chr c) bs = return (Chr c, bs)
dec (e + _) (0b : bs) = do
  (t, bs1) ← dec e bs
  return (InL t, bs1)
dec (_ + e′) (1b : bs) = do
  (t′, bs1) ← dec e′ bs
  return (InR t′, bs1)
dec (e • e′) bs = do
  (t, bs1) ← dec e bs
  (t′, bs′) ← dec e′ bs1
  return (t • t′, bs′)
dec (Star e) bs = do
  (ts, bs′) ← decodeList e bs
  return (List ts, bs′)
dec _ _ = fail "invalid bit code"
```

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.

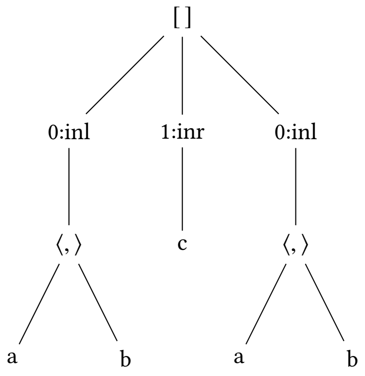
```
decodeList :: Regex → Code → Maybe ([Tree], Code)
decodeList _ [] = fail "fail decodeList"
decodeList _ (1_b : bs) = return ([], bs)
decodeList e (0_b : bs) = do
  (t, be) ← dec e bs
  (ts, bs′) ← decodeList e be
  return (t : ts, bs′)
```

Function decodeList generate a list of parse trees consuming the bit 0_b used as a separator, and bit 1_b which finish the list of parsing results for star operator.

Finally, using dec, the definition of decode is immediate.

```
decode :: Regex → Code → Maybe Tree
decode e bs
   = case dec e bs of
    Just (t, []) → Just t
    _ → Nothing
```

Example 2. We present again the same RE and string we showed in Example 1, 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 0001001. 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 are specified by the next theorem.

Theorem 2. *Let t be a parse tree such that $\vdash t : e$, for some RE e. Then* (code t e) $\triangleright e$ *and* decode e (code t e) = Just t.

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

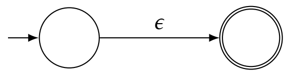
1. Case rule $T1$: Then, $e = \epsilon$ and t = (). Conclusion follows by rule $B1$ and the definition of functions code and decode.
2. Case rule $T2$: Then, $e = a$ and t = Chr a. Conclusion follows by rule $B2$ and the definition of functions code and decode.
3. Case rule $T3$: Then, $e = e1 + e2$ and t = InL tl. By the induction hypothesis, we have that code tl e1 = bs, $bs \triangleright e1$ and decode e1 bs = Just tl. From the definition of code, we have that code (InL tl) (e1 + e2) = 0_b : bs and by rule $B3$, we have that $0_b : bs \triangleright (e1 + e2)$. The conclusion follows from the definition of code, decode and the fact that decode e1 bs = Just tl.
4. Case rule $T4$: Then, $e = e1 + e2$ and t = InR tr. By the induction hypothesis, we have that code tr e2 = bs, $bs \triangleright e2$ and decode e2 bs = Just tr. From the definition of code, we have that code (InR tl) (e1 + e2) = 1_b : bs and by rule $B4$, we have that $1_b : bs \triangleright (e1 + e2)$. The conclusion follows from the definition of code, decode and the fact that decode e2 bs = Just tr.
5. Case rule $T5$: Then, $e = e_1\, e_2$ and t = tl • tr. Conclusion follows from the induction hypothesis on tl and tr.
6. Case rule $T6$: Then, $e = e_1^\star$ and t = List ts, where $\forall t'.t' \in ts \rightarrow\vdash t' : e_1$. The desired conclusion follows from the induction hypothesis on each tree $t' \in ts$.

□

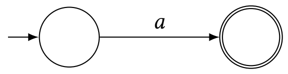
Next, we review Thompson NFA construction which is similar to the proposed semantics for RE parsing developed in Section 3.

2.3 Thompson NFA construction

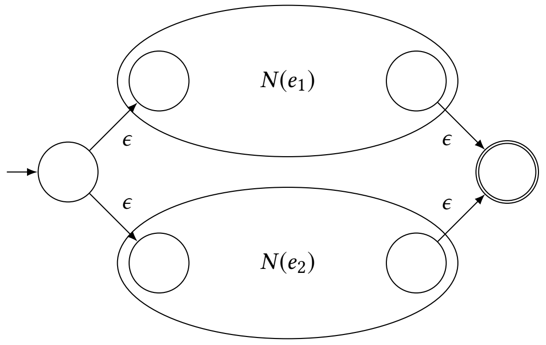
The Thompson NFA construction is a classical algorithm for building an equivalent NFA with ϵ-transitions by induction over the structure of an input RE. We follow a presentation given in [2] where $N(e)$ denotes the NFA equivalent to RE e. The construction proceeds as follows. If $e = \epsilon$, we can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



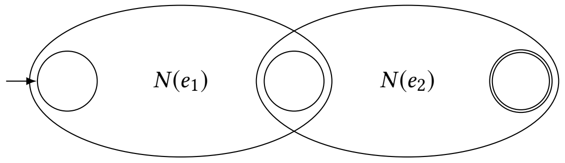
If $e = a$, for $a \in \Sigma$, we can make a NFA with a single transition consuming a:



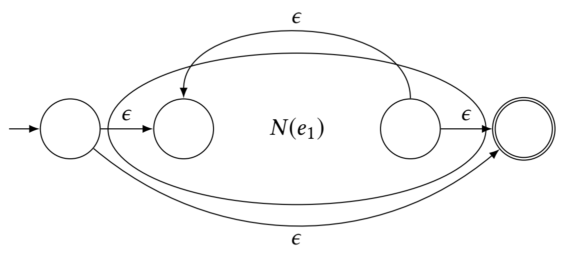
When $e = e_1 + e_2$, we let $N(e_1)$ be the NFA for e_1 and $N(e_2)$ the NFA for e_2. The NFA for $e_1 + e_2$ is built by adding a new initial and accepting state which can be combined with $N(e_1)$ and $N(e_2)$ using ϵ-transitions as shown in the next picture.



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

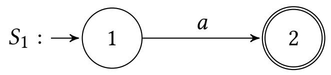


Finally, for the Kleene star operator, we built a NFA for the RE e, add a new starting and accepting states and the necessary ϵ transitions, as shown below.

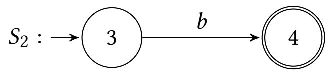


Example 3. In order to show a step-by-step automaton construction following Thompson's algorithm, we take as example the RE $((ab) + c)^*$ over the alphabet $\Sigma = \{a, b, c\}$.

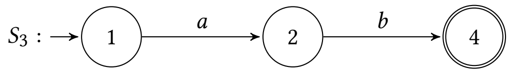
The first step is to construct an automaton (S_1) that accepts the symbol a.



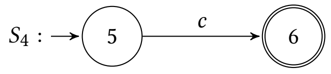
Then, we construct another automaton (S_2) that accepts the symbol b:



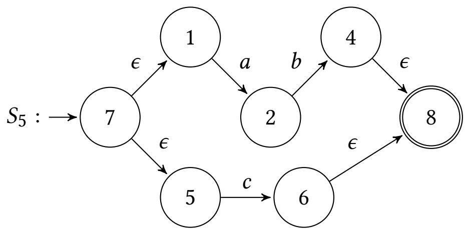
The concatenation ab is accepted by automaton S_3:



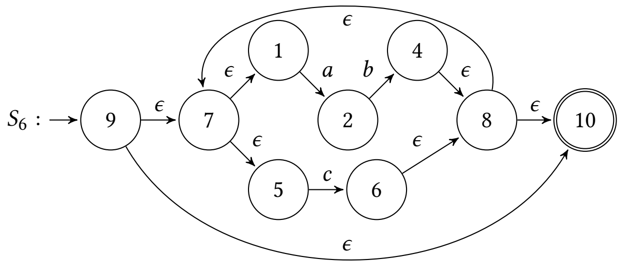
Now, we build automaton S_4, which recognizes the symbol c:



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



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



Originally, Thompson formulate its construction as a IBM 7094 program [31]. In our work, we reformulate it as a small-step operational semantics using contexts, modeled as data-type derivatives for RE, which is the subject of the next section.

2.4 Data-type derivatives

The usage of evaluation contexts is standard in reduction semantics [10]. Contexts for evaluating a RE during the parse of a string s can be defined by the following context-free syntax:

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

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 semantics (Figure 4), 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 its coworkers [23] 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:

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

Constructor InChoiceL store 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 subexpresssions that need to be evaluated during input string parsing. Finally, InStar marks that we are currently processing an expression with a Kleene star operator.

3 Proposed 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 1) 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 [27].

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 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 leading to a new state $\langle d', e', c', b', s'\rangle$ using the rules specified in Figure 4.

The rules of the semantics 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-automata of Thompson NFA and finishing rules to transitions exiting a sub-automata.

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

$$\frac{}{\langle B, \epsilon, c, b, s\rangle \longrightarrow \langle F, \epsilon, c, b, s\rangle}\ (Eps) \qquad \frac{}{\langle B, a, c, b, a : s\rangle \longrightarrow \langle F, a, c, b, s\rangle}\ (Chr) \qquad \frac{\begin{array}{c} b' = 0_b : b \\ c' = E[\,] + e' : c \end{array}}{\langle B, e + e', c, b, s\rangle \longrightarrow \langle B, e, c', b', s\rangle}\ (Left_B)$$

$$\frac{\begin{array}{c} b' = 1_b : b \\ c' = e + E[\,] : c \end{array}}{\langle B, e + e', c, b, s\rangle \longrightarrow \langle B, e', c', b', s\rangle}\ (Right_B) \qquad \frac{c' = E[\,]e' : c}{\langle B, ee', c, b, s\rangle \longrightarrow \langle B, e, c', b, s\rangle}\ (Cat_B) \qquad \frac{}{\langle B, e^\star, c, b, s\rangle \longrightarrow \langle B, e, \star : c, 0_b : b, s\rangle}\ (Star_1)$$

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

$$\frac{c = E[\,] + e' : c'}{\langle F, e, c, b, s\rangle \longrightarrow \langle F, e + e', c', 0_b : b, s\rangle}\ (Left_E) \qquad \frac{c = e + E[\,] : c'}{\langle F, e, c, b, s\rangle \longrightarrow \langle F, e + e', c', 1_b : b, s\rangle}\ (Right_E)$$

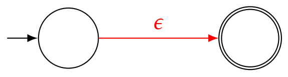
$$\frac{}{\langle F, e, \star : c, b, s\rangle \longrightarrow \langle B, e, \star : c, 0_b : b, s\rangle}\ (Star_{E1}) \qquad \frac{}{\langle F, e, \star : c, b, s\rangle \longrightarrow \langle F, e^\star, c, 1_b : b, s\rangle}\ (Star_{E2})$$

Figure 4. Small-step semantics for RE parsing.

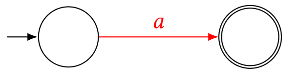
starting configuration with RE ϵ. We can finish any configuration for RE Chr a if it is starting with current string with a leading a. Whenever we have a starting configuration with a choice RE, $e_1 + e_2$, we can non-deterministically choose if input string s can be processed by e_1 (rule $Left_B$) or e_2 (rule $Right_B$). For beginning configurations with concatenation, we parse 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, 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 1_b to end its matching list or restart with RE e for another matching over the input string.

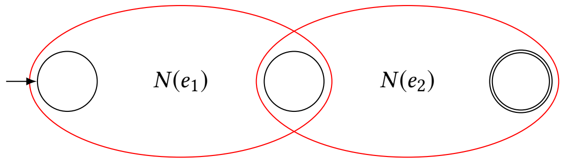
The proposed semantics is inspired by Thompson's NFA construction (as shown in Section 2.3). First, the rule *Eps* can be understood as executing the transition highlighted in red in the following schematic automata.



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

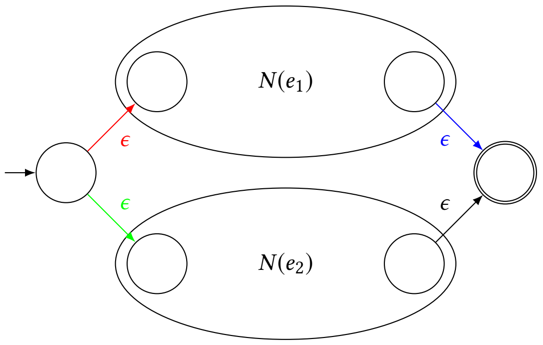


Rule Cat_B corresponds to start the processing of the input string in the automata $N(e_1)$; while rule Cat_{EL} deals with exiting the automata $N(e_1)$ followed by processing the remaining string in $N(e_2)$. Rule Cat_{ER} deals with ending the processing in the automata below.

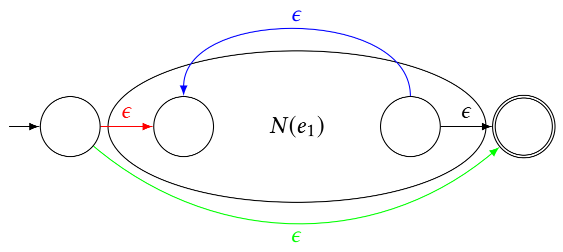


If we consider a RE $e = e_1 + e_2$ and lets $N(e_1)$ and $N(e_2)$ be two NFAs for e_1 and e_2, respectively, we have the following correspondence between transtions and semantics rules in the next NFA:

- 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 automata 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 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 4. 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 1).

Theorem 3. *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 $\langle B, e_1, ctx, b, s\rangle \rightarrow^\star \langle E, e', ctx', b', []\rangle$ 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 $\langle B, e_2, ctx, b, s\rangle \rightarrow^\star \langle E, e', ctx', b', []\rangle$ 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 $\langle B, e_1, ctx, b, s\rangle \rightarrow^\star \langle E, e', E[\,]\, e_2 : ctx, b', []\rangle$ and by induction hypothesis on $s_2 \in [\![e_2]\!]$, we have $\langle B, e_2, e_1\, E[\,] : ctx, b, s\rangle \rightarrow^\star \langle E, e', ctx, b', []\rangle$ 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 $\langle B, e_1, \star : ctx, b, s_1\rangle \rightarrow^\star \langle E, e', \star : ctx, b', []\rangle$, the induction hypothesis on $s_2 \in [\![e_1^\star]\!]$ give us $\langle B, e_1^\star, \star : ctx, b, s_2\rangle \rightarrow^\star \langle E, e', \star : ctx, b', []\rangle$ and conclusion follows.

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

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

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

   By induction hypothesis on e_1 and the derivation $\langle B, e_1, E[\,]+e_2 : ctx, b, s\rangle \rightarrow^\star \langle E, e', ctx', b', []\rangle$ 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:

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

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

□

4 Implementation details

In order to implement the small-step semantics of Figure 4, 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 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. 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 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.

4.1 Test suite

An overview of QuickCheck. Our tests are implemented using QuickCheck [6], 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]
```

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

Generation of random RE 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 give 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 generate REs that do not contain sub-REs of the form $e^\star$, where e is nullable[3]. All results can be extended to problematic[4] REs in the style of Frisch et. al [14].

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 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. 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. We use the accept function from [12] for this and compare 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 4. *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 t = () and conclusion follows.
2. Case rule $T2$: Then, $e = a$ and t = Chr a and conclusion follows.
3. Case rule $T3$: Then, $e = e_1 + e_2$ and 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 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 t = tl • tr. Conclusion is immediate from the induction hypothesis.
6. Case rule $T6$: Then, $e = e_1^\star$ and 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, t = () and the conclusions follows by rule $T1$.
2. Case $e = a$: Then, 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 t = InL tl: By induction hypothesis, we have that tc tl e1 = True and conclusion follows.
   b. Case t = InR tr: By induction hypothesis, we have that tc tr e2 = True and conclusion follows.
4. Case $e = e_1\,e_2$: Then, t = tl • tr and conclusion follows by the induction hypothesis and the rule $T5$.
5. Case $e = e_1^\star$: Then, t = List ts and conclusion follows by induction hypothesis on each element of ts and rule $T6$.

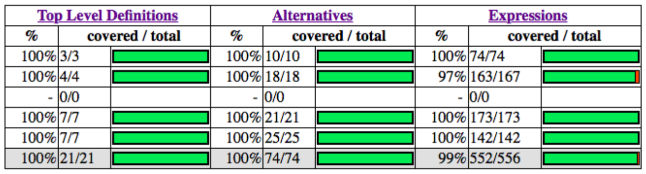
□

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) [15] to generate statistics about the execution of our tests. Code coverage results are presented in Figure 5.

Our test suite give us almost 100% of code coverage, which provides a strong evidence that our semantics is indeed correct. All top

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

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



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 5. Code coverage results

level definitions and function alternatives are actually executed by the test cases and just two expressions are marked as non-executed by HPC.

4.2 Coq formalization

In this section we briefly present the status of our Coq formalization. First, we give a suscint introduction to Coq focusing on features used in our formalization and next we describe the implementation of our verified interpreter for the proposed semantics (Figure 4).

4.2.1 A tour of Coq proof assistant

Coq is a proof assistant based on the calculus of inductive constructions (CIC) [4], a higher-order typed λ-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called "BHK-correspondence"[5], where 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) [29].

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 in the construction of proof terms.

In this section we provide a brief overview of Coq. We start with the 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, that indicates that it belongs to the Set sort[6]. Type nat is formed by two data constructors: O, that represents the number 0, and S, the successor function.

```
Inductive nat : Set :=
 | O : nat
 | S : nat → nat.
Fixpoint plus (n m : nat) : nat :=
  match n with
   | O ⇒ m
   | S n′ ⇒ S (plus n′ m)
  end.
Theorem plus_0_r : ∀ n, plus n 0 = n.
Proof.
  intros n.induction n as [| n′].
  reflexivity.
  simpl.rewrite → IHn′.reflexivity.
Qed.
```

Command **Fixpoint** allows the definition of 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 exemple, we show a simple theorem about plus, that 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 an interactive section of Coq, after enunciation of theorem plusOr, we must prove the following goal:

```
 =====================
  ∀ n : nat, plus n 0 = n
```

After command **Proof**, we can use tactics to build, step by step, a term of the given type. The first tactic, intros, is used to move premisses and universally quantified variables from the goal to the hypothesis. As a result of using intros, the quantified variable n is moved from the goal to the hypothesis, resulting in the following:

```
n : nat
 =====================
plus n 0 = n
```

In order to prove goal plus n 0 = n we can proceed by induction over the structure of n, by using tactic induction. This generates one goal for each constructor of type nat (O and S), leaving us with two goals to be proved:

```
2 subgoals
 =====================
  plus 0 0 = 0
subgoal 2 is :
plus (S n′) 0 = S n′
```

Goal plus 0 0 = 0 can be shown to hold directly by the definition of plus. Tactic reflexivity proves such equalities, after reducing both sides of the equality to their normal forms. The next goal to be proved is:

```
n′ : nat
IHn′ : plus n′ 0 = n′
 =====================
  plus (S n′) 0 = S n′
```

Tactic induction automatically generates the induction hypothesis IHn′ for this theorem. In order to finish the proof, we need to transform the goal to use the inductive hypothesis. In this case we can simply use tactic simpl, for performing reductions based on the definition of plus. This changes the goal to:

```
n′ : nat
IHn′ : plus n′ 0 = n′
 =====================
  S (plus n′ 0) = S n′
```

Since the goal now has exactly the left hand side of the hypothesis IHn′ as a sub-term , we can use tactic rewrite. The use of rewrite →

[5]Abbreviation of Brower, Heyting, Kolmogorov, de Bruijn and Martin-Löf Correspondence. This is also known as the Curry-Howard "isomorphism".
[6]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.

```
λ n : nat ⇒
  nat_ind
  (λ n0 : nat ⇒ plus n0 O = n0) (eq_refl O)
    (λ (n′ : nat) (IHn′ : plus n′ O = n′) ⇒
      eq_ind_r (λ n0 : nat ⇒ S n0 = S n′)
        (eq_refl (S n′)) IHn′) n
          : ∀ n : nat, plus n O = n
```

Figure 6. Term that represents the proof of theorem plus_0_r.

IHn′ replaces plus n′ 0 by n′ (rewrite ← IHn′ does the reverse, i.e. replaces n′ by plus n′ 0). We obtain now the following:

```
n′ : nat
IHn′ : plus n′ 0 = n′
 =====================
  S n′ = S n′
```

Goal S n′ = S n′ can be proven by using reflexivity.

This tactic script creates the term presented in Figure 6. For each inductively defined data type, Coq generates automatically an induction principle [4, Chapter 14]. For natural numbers, the following Coq term, called nat_ind, is created:

```
nat_ind
  : ∀ P : nat → Prop,
    P O → (∀ n : nat, P n → P (S n)) →
    ∀ 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 (a value of type ∀ n:nat, P n → P (S n)). Besides nat_ind, generated by the use of tactic induction, the term in Figure 6 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. 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 : ∀ 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 of type 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:

∀ n : nat, n≠O → { p | n = S p }.

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 in the definition of 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 pred_cert : ∀ n : nat, n≠O → { p | n = S p }.
  intros n H.
  destruct n as [| 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 = O }

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 (knowns 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 pred_cert definition.

```
Definition pred_cert : ∀ n : nat, { p | n = S p } + { n = 0 }.
  refine (λ n ⇒
    match n with
     | O ⇒ inright _
     | S n′ ⇒ inleft _ (exist _ n′ _)
    end); reflexivity.
Defined.
```

4.2.2 A verified interpreter for the small-step semantics

RE Syntax and contexts definition. We let the following inductive type denote the syntax of RE.

```
Inductive regex : Set :=
 | Emp : regex
 | Eps : regex
 | Chr : ascii → regex
 | Cat : regex → regex → regex
 | Choice : regex → regex → regex
 | Star : regex → regex.
```

To ease the task of writing code that manipulate regex values, we introduce notations to represent its constructor in a more friendly way. For exemple, let e, e1 : regex. Instead of writing Cat e1 e2, after defining the notations below, we can just write e1@e2.

```
Notation "'#0'" := Emp.
Notation "'#1'" := Eps.
Notation "'$' c" := (Chr c) (at level 40).
Notation "e '@' e1" := (Cat e e1)
  (at level 15, left associativity).
Notation "e ':+:' e1" := (Choice e e1)
  (at level 20, left associativity).
Notation "e '^*'" := (Star e) (at level 40).
```

Having defined the RE syntax, our next step is to define its data-type of one-hole contexts.

```
Inductive hole : Set :=
 | LeftChoiceHole : regex → hole
 | RightChoiceHole : regex → hole
 | LeftCatHole : regex → hole
 | RightCatHole : regex → hole
 | StarHole : hole.
Definition ctx := list hole.
```

The structure of hole and regex are just translations of its Haskell's correspondent definitions to Coq.

Small-step semantics. Unlike Haskell, Coq supports full dependent types which allow us to specify inductively defined relations as Coq types. Representing our small-step semantics in Coq is just a matter of defining a data type such that its constructors denote the rules of the proposed semantics. Data type dir denotes the direction and conf the configuration used by the semantics.

```
Inductive dir : Set :=
 | B : dir | F : dir.
Inductive conf : Set :=
  MkConf : dir → regex → ctx → list bit → string → conf.
Notation "[[ d , e , c , b , s ]]" := (MkConf d e c b s).
```

As usual in Coq's developments, we use a notation to ease the task of writing conf values.

```
Inductive step : conf → conf → Prop :=
 | Eps : ∀ c b s,
   [[B, #1, c, b, s]]⇒[[F, #1, c, b, s]]
 | Chr : ∀ c b s a,
   [[B, $a, c, b, (String a s)]]⇒[[F, $a, c, b, s]]
 | Left_B : ∀ c b s e e′ c′ b′,
   c′ = LeftChoiceHole e′ :: c →
   b′ = 0_b :: b →
   [[B, e + e′, c, b, s]]⇒[[B, e, c′, b′, s]]
 | Left_E : ∀ e e′ c s b c′ b′,
   c′ = LeftChoiceHole e′ :: c →
   b′ = 0_b :: b →
   [[F, e, c′, b, s]]⇒[[F, e + e′, c, b′, s]]
(∗some constructors omitted∗)
where c⇒c1 := (step c c1).
```

Interpreter definition. Having defined our semantics as a Coq data type, implementing its certified interpreter is just a matter of expressing its correctness property as the interpreter's type.

Our first step in the formalization is to define a verified version of next function which produces a list of configurations reached after executing one-step from a given input Conf. The Coq type for next is as follows:

```
Definition next (c : conf) : {cs : list conf | ∀ c′, In c′ cs → c⇒c′}.
```

Such type specifies that from a given configuration (c : conf) we produce a list cs of configurations that can be reached in one step from c (i.e. such that for any c′∈ cs, we have that c⇒c′). Since such function definition involve proof terms, we use the refine tactic to mark proof positions as holes to be filled latter with tactics. Below, we show parts of the next definition.

```
refine (match c with
  | [[B, #1, co, b, s]] ⇒
    exist _ ([[F, #1, co, b, s]]::nil) _
  | [[B, $a, co, b, (String a′ s)]] ⇒
    if ascii_dec a a′ then
      exist _ ([[F, $a, co, b, s]]::nil) _
    else exist _ (@nil conf) _
  | [[B, e + e′, c, b, s]] ⇒
    exist _ ([[B, e, LeftChoiceHole e′::c, 0_b::b, s]]::
      [[B, e′, RightChoiceHole e::c, 1_b::b, s]]::nil) _
  (∗some code omitted∗)
 end) (∗tactics omitted∗)
```

The first equation of next definition shows the code for the semantics rule *Eps*: Given a configuration [[B, #1, co, b, s]], the unique possible result is a singleton list containing [[F, #1, co, b, s]]. Second equation deals with single character REs and it specifies that if the input string starts with the same character as the RE's, the result should be a list containing only the configuration [[F, $a, co, b, s]]; otherwise an empty list is returned. Whenever the input conf is of the form [[B, e + e′, c, b, s]], we need to output the two possible resulting configurations: one for the left-hand side of the input RE:

[[B, e, LeftChoiceHole e′::c, 0_b::b, s]]

which corresponds to rule $Left_B$ and the configuration equivalent to rule $Right_B$

[[B, e′, RightChoiceHole e::c, 1_b::b, s]]

The equations for other rules follows a similar pattern and are omited for brevity.

Having defined the function next, we need to compute its reflexive-transtive closure. We let notation $\Rightarrow^\star$ denote the reflexive-transitive closure of the relation step. In order to build the closure of next function, we use some auxiliar definitions.

First, we define nexts_type as a type for a function that returns a list of all accessible configurations, in one-step of execution, from a given input list of conf's.

```
Definition nexts_type (cs : list conf) :=
  {cs′ | ∀ c′, In c′ cs′ → ∃ c, In c cs∧c⇒c′} + {cs = []}.
```

We name such function as nexts and we omit its refine tactic based definition. Finally, our certified interpreter is built using nexts function and its type is presented below:

Definition steps : ∀ (cs : list conf) (n : nat),
{css | ∀ c′, In c′ css → ∃ c, In c cs∧$c \Rightarrow^{\star} c'$} + {cs = [ ]}.

Again, we use refine tactic to construct such function. Its type specifies that it returns a list configurations that can be reached after zero or more steps from the input configuration list. It is worth to mention that such function is defined by recursion over a "fuel" parameter [24] to ensure its structurally recursive definition. Currently, we are working on defining steps function using well-founded relations or even domain predicates [5].

5 Related work

Ierusalimschy [17] proposed the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. He argued that pure REs is a weak formalism for pattern-matching tasks: many interesting patterns either are difficult to 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 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. In [25], Medeiros et. al. presents informal correctness proofs of LPEG PM. While such proofs represent a important step towards the correctness of LPEG, there is no guarantee that LPEG implementation follows its specification.

In [27], Rathnayake and Thielecke 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 parsing of REs), they did not present any evidence or proofs that their VM is correct.

Fischer, Huch and Wilke [12] 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 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 proofs of their Haskell code.

Cox [7] 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 is even mentioned.

Frisch and Cardelli [14] 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 identify three notions of problematic words, REs, and values (which represent the ways to match words), relate these three notions, and propose matching algorithms to deal with the problematic case.

Ribeiro and Du Bois [28] 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 provide formal proofs, their tool show a bad performance when compared with other approaches to RE parsing. Besides, they did not prove that their algorithm follows some disambiguation policy, like POSIX or greedy.

Nielsen and Henglein [26] 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 [9] to emit a bit representation without explicitly materializing the parse tree itself. They also showed that Frisch and Cardellifis greedy RE parsing algorithm [14] 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.

An algorithm for POSIX RE parsing is described in [30]. The main idea of the article is to adapt derivative parsing 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. use a bit encoded representation of RE parse trees. Textual proofs of correctness of the proposed algorithm are presented in an appendix.

6 Conclusion

In this work, we presented a small-step operational semantics for a virtual machine for RE parsing inspired on Thompson's NFA construction. Our semantics produces, as parsing evidence, bit-codes which can be used to characterize which disambiguation strategy is followed by the semantics. We use data-type derivatives to represent evaluation contexts for RE. Such contexts are used to decide how to finish the execution of the RE on focus. We have developed a prototype implementation of our semantics in Haskell and use QuickCheck to verify its relevant properties with respect to a simple implementation of RE parsing by Fisher et. al. [12].

Currently, we have a formalized interpreter of our semantics in Coq proof assistant [4] available at project's on-line repository [8]. We are working on formalizing the equivalence between the proposed semantics and the standard RE inductive semantics.

As future work we intend to use our verified semantics to build a certified tool for RE parsing, work on proofs that the semantics follow a specific disambiguation strategy and investigate how other algorithms (e.g. the Glushkov construction [16]) for converting a RE into a finite state machine could be expressed in terms of an operational semantics.

References

[1] Michael Gordon Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2003. Derivatives of Containers. In *Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings. (Lecture Notes in Computer Science)*, Martin Hofmann (Ed.), Vol. 2701. Springer, 16–30. https://doi.org/10.1007/3-540-44904-3_2

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

[3] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. 2010. Regular Expressions, au point. *CoRR* abs/1010.2604 (2010). arXiv:1010.2604 http://arxiv.org/abs/1010.2604

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

[5] Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science* 26, 1 (2016), 38–88. https://doi.org/10.1017/S0960129514000115

[6] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00).* ACM, New York, NY, USA, 268–279.

[7] Russ Cox. 2009. Regular Expression Matching: the Virtual Machine Approach. (2009). https://swtch.com/{~}rsc/regexp/regexp2.html

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

[9] Danny Dubé and Marc Feeley. 2000. Efficiently Building a Parse Tree from a Regular Expression. *Acta Inf.* 37, 2 (Oct. 2000), 121–144. https://doi.org/10.1007/s002360000037

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

[11] Denis Firsov and Tarmo Uustalu. 2013. Certified Parsing of Regular Languages. In *Proceedings of the Third International Conference on Certified Programs and Proofs - Volume 8307.* Springer-Verlag New York, Inc., New York, NY, USA, 98–113. https://doi.org/10.1007/978-3-319-03545-1_7

[12] Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. A play on regular expressions. *ACM SIGPLAN Notices* 45, 9 (2010), 357. https://doi.org/10.1145/1932681.1863594

[13] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04).* ACM, New York, NY, USA, 111–122. https://doi.org/10.1145/964001.964011

[14] Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. *ICALP 2004 - International Colloquium on Automata, Languages and Programming* 3142 (2004), 618–629.

[15] Andy Gill and Colin Runciman. 2007. Haskell Program Coverage. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07).* ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1291201.1291203

[16] Victor M. Glushkov. 1961. The Abstract Theory of Automata. *Russian Mathematical Surveys* 16, 5 (1961), 1–53.

[17] Roberto Ierusalimschy. 2009. A text patternmatching tool based on parsing expression grammars. *Software - Practice and Experience* (2009). https://doi.org/10.1002/spe.892

[18] Donald E. Knuth. 1971. Top-down Syntax Analysis. *Acta Inf.* 1, 2 (June 1971), 79–110. https://doi.org/10.1007/BF00289517

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

[20] A. Loh. [n. d.]. Typesetting Haskell and more with lhs2TeX. ([n. d.]). http://www.cs.uu.nl/~{}andres/lhs2TeX-IFIP.pdf

[21] Raul Lopes, Rodrigo Ribeiro, and Carlos Camarão. 2016. Certified Derivative-Based Parsing of Regular Expressions. In *Programming Languages — Lecture Notes in Computer Science 9889.* Springer, 95–109.

[22] Raul Felipe Pimenta Lopes. 2018. Certi ed Derivative-based Parsing of Regular Expressions. (2018).

[23] Conor McBride. 2008. 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.* 287–295. https://doi.org/10.1145/1328438.1328474

[24] Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings (Lecture Notes in Computer Science)*, Ralf Hinze and Janis Voigtländer (Eds.), Vol. 9129. Springer, 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

[25] Sérgio Medeiros and Roberto Ierusalimschy. 2008. A parsing machine for PEGs. *Proceedings of the 2008 symposium on Dynamic languages - DLS '08* (2008), 1–12. https://doi.org/10.1145/1408681.1408683

[26] Lasse Nielsen and Fritz Henglein. 2011. Bit-coded Regular Expression Parsing. In *Language and Automata Theory and Applications*, Adrian-Horia Dediu, Shunsuke Inenaga, and Carlos Martín-Vide (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 402–413.

[27] Asiri Rathnayake and Hayo Thielecke. 2011. Regular Expression Matching and Operational Semantics. *Electronic Proceedings in Theoretical Computer Science* 62, Sos (2011), 31–45. https://doi.org/10.4204/EPTCS.62.3 arXiv:1108.3126

[28] Rodrigo Ribeiro and André Du Bois. 2017. Certified Bit-Coded Regular Expression Parsing. *Proceedings of the 21st Brazilian Symposium on Programming Languages - SBLP 2017* (2017), 1–8. http://dl.acm.org/citation.cfm?doid=3125374.3125381

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

[30] Martin Sulzmann and Kenny Zhuo Ming Lu. 2014. POSIX Regular Expression Parsing with Derivatives. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 203–220.

[31] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422.

A Correctness of the accept function

Fisher et. al. [12] presents a simple and elegant function for parsing a string using a RE. It relies on two auxiliar 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 1. *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 2. *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 5. *For all* s *and* e, accept e s ≡ True *if, and only if,* $s \in [\![e]\!]$.

Proof.

- ($\rightarrow$) : By induction on the structure of e using lemmas about parts and split.
- ($\leftarrow$) : By induction on the derivation of $s \in [\![e]\!]$.

□