Elsevier Editorial System(tm) for Science of Computer Programming

Manuscript Draft

Manuscript Number:

Title: Certified Virtual Machine Based Regular Expression Parsing

Article Type: VSI: SBLP 2018

Keywords: Regular Expressions; Virtual Machines; Parsing; Coq proof assistant

Corresponding Author: Mr. Rodrigo Geraldo Ribeiro, Ph.D.

Corresponding Author's Institution: Universidade Federal de Ouro Preto

First Author: Thales Delfino

Order of Authors: Thales Delfino; Rodrigo Geraldo Ribeiro, Ph.D.; Andre R Du Bois, Ph.D.; Elton Cardoso; Leonardo Reis, Ph.D.

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 REs to their corresponding finite state machines 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 generating 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 show that the proposed semantics is sound and complete w.r.t. a standard inductive semantics for RE and that the evidence produced by it denotes a valid parsing result. All of our results are formalized in Coq proof assistant and from it we extract a certified algorithm which we use to build a RE parsing tool using Haskell programming language. Experiments comparing the efficiency of our algorithm with other approaches implemented using Haskell are reported.

Highlights (for review)

- An operational semantics for a virtual machine for regular expression parsing.
- The semantics produces, as evidence, correct bit-coded parse trees.
- A correct algorithm to convert problematic regular expressions into non-problematic.
- Certified interpreter for the semantics in Coq proof assistant.

Certified Virtual Machine Based Regular Expression Parsing

Thales Delfino, Rodrigo Ribeiro

Programa de Pós-Graduação em Ciência da Computação - PPGCC, Universidade Federal de Ouro Preto

André Rauber Du Bois

Programa de Pós-Graduação em Computação - PPGC, Universidade Federal de Pelotas

Elton Cardoso

Departamento de Computação e Sistemas - DECSI, Universidade Federal de Ouro Preto

Leonardo Reis

Departamento de Computação, Universidade Federal de Juiz de Fora

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 REs to their corresponding finite state machines 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 generating 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 show that the proposed semantics is sound and complete w.r.t. a standard inductive semantics for RE and that the evidence produced by it denotes a valid parsing result. All of our results are formalized in Coq proof assistant and from it we extract a certified algorithm which we use to build a RE parsing tool using Haskell programming language. Experiments comparing the efficiency of our algorithm with other approaches implemented using Haskell are reported.

Keywords: Regular Expressions, Virtual Machines, Parsing, Coq proof

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 [1]. 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 [2, 3, 4, 5]).

Approaches for RE parsing can use representations of finite state machines (e.g. [2]), derivatives (e.g. [3, 6, 4]) or the so-called pointed RE's or its variants [5, 7]. 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 [8]. Recently, some works have tried to revive the use of such machines for parsing: Cox [9] 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 [10] defines a VM whose instruction set can be used to compile Parser Expressions Grammars (PEGs) [11]. 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 guarantee is even mentioned. Second, it does not provide an evidence for matching, which could be used to characterize a disambiguation strategy, like Greedy [1] and POSIX [12]. 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 [13], 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 develops its VM using a instruction set like syntax and semantics, we use, as inspiration, virtual machines for the λ-calculus, like the SECD and Krivine machines [14, 15].

One important issue regarding RE parsing is how to deal with the so-called problematic RE[1] [1]. In order to avoid the well-known issues with problematic RE, we use a transformation proposed by Medeiros et. al. [16] which turns a problematic RE into an equivalent non-problematic one. We prove that this algorithm indeed produce equivalent REs using Coq proof assistant.

Our contributions are:

- We present a big step semantics for RE parsing VM which produces bit-codes as parsing evidence.

[1] We say that a RE e is problematic if there's exists e_1 s.t. $e = e_1^\star$ and e_1 accepts the empty string.

- We develop a certified implementation of an algorithm that converts a problematic RE into a non-problematic one.
- We prove that the bit-codes produced by our VM are valid parsing evidence.
- We extract from our formalization a certified algorithm in Haskell and use it to build a RE parsing tool. We compare its performance against well known Haskell library for RE parsing.

This paper describes the continuation of the RE VM-based parsing research which we previously reported on a paper published on SBLP 2018 [17]. The current work improves our previous results mainly by: 1) using a big-step operational semantics which deals correctly with problematic REs, unlike our previous small-step semantics. We simply transform problematic REs into equivalent non-problematic ones before starting their execution by our semantics; 2) all results of this paper are completely mechanized using Coq proof assistant. Our previous work used property-based testing in order to provide an evidence for the correctness of the small-step semantics.

The rest of this paper is organized as follows. Section 2 gives a suscint introduction to Coq proof assistant. Section 3 presents some background concepts on RE and its parsing problem. Our operational semantics for RE parsing and its theoretical properties are described in Section 4. Our Coq formalization is described in Section 5. Section 6 presents some experimental results regarding the tool produced using the verified algorithm and Section 7 discuss related works. Finally, Section 8 concludes and presents some directions for future work.

While all the code on which this paper is based has been developed in Coq, we adopt a "lighter" syntax when presenting its code fragments. We chose this presentation style in order to improve readability, because functions that use dependently typed pattern matching require a high number of type annotations, which would deviate from our objective of providing an easily understandable formalization. For theorems and lemmas, we sketch the proof strategy but omit tactic scripts.

Coq formalization. All source code produced, including the source of this article, instructions on how to build it and replicate the reported experiments are avaliable on-line [18].

2. A tour of Coq proof assistant

Coq is a proof assistant based on the calculus of inductive constructions (CIC) [19], a higher-order typed λ-calculus extended with inductive definitions. Theorem proving in Coq follows the ideas of the so-called "BHK-correspondence"[2], 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) [20].

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 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, that indicates that it belongs to the `Set` sort[3]. 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.
```

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

[3] 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.

```
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 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 example, 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 the example, various tactics are used to prove the desired result. The first tactic, `intros`, is used to move premisses 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 in-

duction principle [19, Chapter 14]. For natural numbers, the following Coq term, called `nat_ind`, is created:

```
nat_ind
     : forall P : nat -> Prop,
       P 0 -> (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 0 = n0) (eq_refl 0)
                   (fun (n' : nat) (IHn' : plus n' 0 = n') =>
                      eq_ind_r (fun n0 : nat => S n0 = S n')
                               (eq_refl (S n')) IHn') n
                : forall n : nat, plus n 0 = 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: $\{x : A \mid P\, x\}$. 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 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 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 (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 `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 suscint way using notations introduced in [21].

```
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 [21].

```
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 5.

A detailed discussion on using Coq is out of the scope of this paper. Good introductions to Coq proof assistant are avaliable elsewhere [19, 21].

3. Background

3.1. 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 Coq formalization, we represent alphabet symbols using type `ascii`. We let concatenation of RE, strings and lists by juxtaposition. Notation $|s|$ denotes the size of an string s. 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 $(\texttt{size}(e), |s|)$.

Following common practice [4, 3, 22], 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 (Figure **??**).

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: $e^\star = \epsilon + e\,e^\star$.

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

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

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

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

Figure 1: RE inductive semantics.

We say that two REs are equivalent, written $e \approx e'$, if the following holds:

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

3.2. 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. and Nielsen et. al. [13, 1], we let parse trees be terms whose type is its underlying RE. The following context-free grammar defines the syntax of parse trees, where we use a Haskell-like syntax for lists.

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

Figure 2: 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 3.

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

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

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

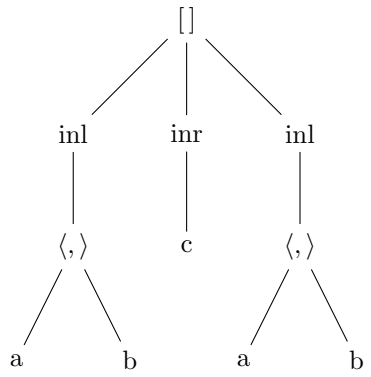
$$\frac{\vdash t : e \quad \vdash ts : e^\star}{\vdash t : ts : e^\star}$$

Figure 3: Parse tree typing relation.

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

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

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 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 1. 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 2. If $\vdash t : e$ then (`flatten` t)$\in [\![e]\!]$.

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

Nielsen et. al. [13] 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. [3] showed an inductively defined relation between valid bit-codes and RE, accordingly to the encoding proposed by [13]. 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 $e\,e'$ 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

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

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

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

Figure 4: Typing relation for bit-codes.

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.

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

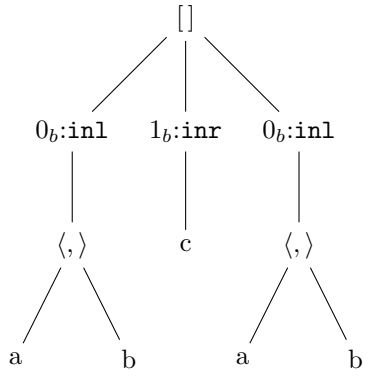
Function `code` has an immediate definition by recursion on the structure of a parse tree. Note that the code generation is driven by input tree's type (i.e. its underlying RE). In the definition of function `decode`, we use an auxiliary function, `decode1`, which threads the remaining bits in recursive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

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

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 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 $0_b0_b0_b1_b0_b0_b1_b$.

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 3. 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 $\vdash t : e$. □

3.3. 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 [1]. Medeiros et al. [16] present a function which converts a problematic RE into a equivalent non-problematic one.

The conversion function relies on two auxiliar 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.

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

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

Functions `nullable` and `empty` obeys 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. [16] 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 sub-expression and $\texttt{f}_{\texttt{in}}$ rewrites the Kleene star subexpression so that it becomes non-problematic and preserves the language of the original RE. The definition of functions $\texttt{f}_{\texttt{in}}$ and $\texttt{f}_{\texttt{out}}$ are presented next.

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

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

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 4. If $\mathtt{f_{out}}(e) = e'$ 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[4]) by Medeiros et. al. [16]. 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 5.

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

4. Proposed semantics for RE parsing

In this section we 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 judgement 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.

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

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

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

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

Figure 5: Operational semantics for RE parsing.

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 $ConsVM$ 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 5 (Soundness)**.** If $\langle e, s\rangle \rightsquigarrow (bs, s_p, s_r)$ then $s = s_p\,s_r$ and $s_p \in [\![e]\!]$.

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

Theorem 6 (Completeness)**.** If $s_p \in [\![e]\!]$ then for all s_r we have that exists bs, s.t. $\langle e, s_p\,s_r\rangle \rightsquigarrow (bs, s_p, s_r)$.

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

Theorem 7 (Parsing result soundness)**.** If $\langle e, s\rangle \rightsquigarrow (bs, s_p, s_r)$ then: 1) $bs \rhd e$; 2) $\texttt{flatten}(\texttt{decode}(bs : e)) = s_p$; and 3) $\texttt{code}(\texttt{decode}(bs : e) : e) = bs$.

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

5. Coq formalization

In this section we describe the main design decisions in our formalization. At the end of this section, 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 1.

```
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 3.1). In order to formalize the needed ordering relation, we take advantage of Coq's standard library, which provide several combinators to assemble well-founded relations. As an example, consider the following fact used by Medeiros et. al [16] to prove the correctness of its $\mathtt{f_{out}}$ 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 proved 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 whether it is 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 [21], which consists in introducing an equality to allow the refinement of each equation type in dependently typed pattern-matching.

In order to specify the emptyness 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 $\mathtt{f}_{\mathtt{out}}$. As pointed by Medeiros et. al. [16], most of the work to produce a unproblematic RE is done by function $\mathtt{f}_{\mathtt{in}}$, 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 $\mathtt{f}_{\mathtt{in}}$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 estabilishes that the return RE `e'` is unproblematic, does not accepts the empty string and that 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 judgement is coded as the following inductive predicate, in which each constructor has a correspondent rule in Figure 3.

```
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 1 and 2 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 4) 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 elaborate recursive definition, as shown in Section 3.2, 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 sub-expressions. 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 semantics definition consists of the Coq representation of the judgement in Figure 5, 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 5 and we omit redundant explanations for brevity.

The soundness and completeness theorems of the proposed 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 a 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 extrac-

tion, which has several pre-defined settings for using data-types and functions of Haskell's Prelude[5].

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 avaliable at project's on-line repository [18].

6. Experimental results

We use the formalized algorithm to build a Haskell tool for RE parsing and compare its performance against the library regex-applicative [24], which is an optimized library for RE matching / parsing for Haskell. The reason for choosing this library is that it allows us to build bit coded parse trees using its applicative interface [25], 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 [12]). The results are presented in Figures 6 and 7.

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 behaviour can be explained by the implementation details of

[5] Prelude is the name of the Haskell library automatically loaded in any Haskell module [23].

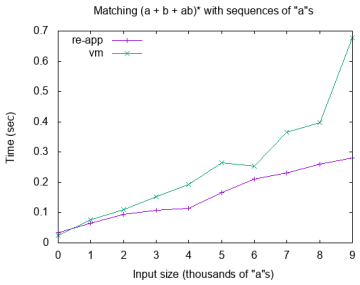


Figure 6: Results of experiment 1.

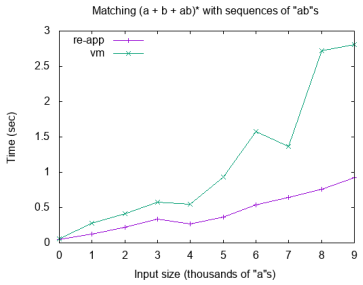


Figure 7: Results of experiment 2.

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 $(a+\epsilon)^n a^n$, where a^n denotes $n \geq 0$ 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 [13]. The results of executing this experiments on increasing values of n is presented below.

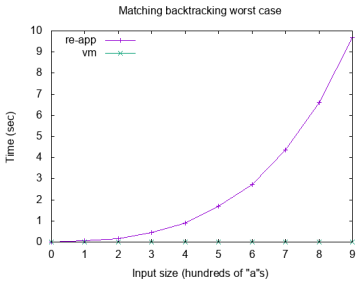


Figure 8: Results of experiment 3.

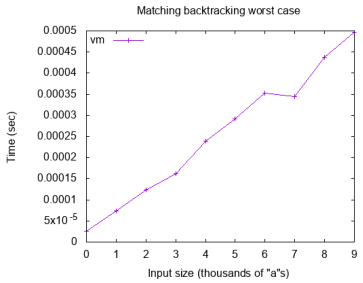


Figure 9: Results of experiment 3 considering only the VM.

In this example, our approach has a much better performance than regex-applicative, which exhibits an exponential behaviour (also known as catastrophic backtracking [26]). Such bad behaviour 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 9.

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 [27]. The experiment consists in collecting the result of running both semantics on thousands of input pairs formed by a RE and strings. The average of such executions is presented in the Figure 10, which shows that both algorithms exhibit a linear behaviour on random inputs.

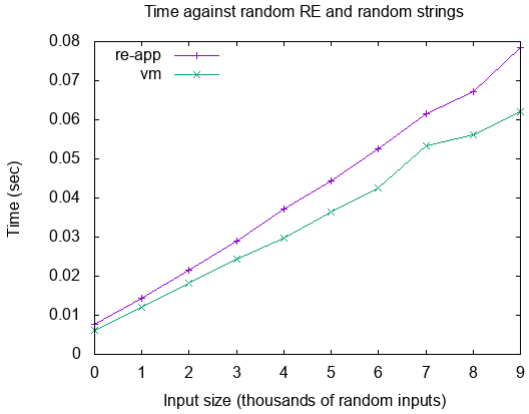


Figure 10: Results of experiment 4.

A few words should be written about how we generate random inputs [6]. 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

[6] 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 avaliable elsewhere [28].

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 Eps), (90, Chr <$> genChar) ]
  | otherwise = frequency [ (10, return Epsilon), (30, Chr <$> genChar)
        , (20, Cat <$> sizedRegex n2 <*> sizedRegex n2)
        , (20, Choice <$> 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')
```

7. Related works

Ierusalimschy [10] proposed the use of Parsing Expression Grammars (PEGs) as a basis for pattern matching. He argued that pure REs are 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 that allows an implementation of PEGs for pattern matching. Medeiros et. al. [29] presents informal correctness proofs of LPEG parsing machine. While such proofs represent an important step towards the correctness of LPEG, there is no guarantee that LPEG implementation follows its specification.

Rathnayake and Thielecke [22] 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 [7] developed a Haskell program for matching REs. Their 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 proof of their Haskell code.

Cox [9] 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 [1] 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 [3] 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 them-

selves, named verigrep, for RE-based search in the style of GNU grep. While the authors provided formal proofs, their tool show a bad performance when compared to other approaches to RE parsing.

Nielsen and Henglein [13] 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 Dub and Feeley [30] to emit a bit representation without explicitly materializing the parse tree itself. They also showed that Frisch and Cardellis greedy RE parsing algorithm [1] 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.

Sulzmann et. al. [12] propose 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. use a bit encoded representation of RE parse trees. Textual proofs of correctness of the proposed algorithm are presented in an appendix. Ausaf et. al. [31] 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.

A recent application of REs was presented by Radanne [32]. 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.

Radanne and Thiemann [33] 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 instance 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.

Groz and Maneth [34] 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.

A formal constructive theory of RLs was presented by Doczkal et. al. in [35]. 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.

A new technique for constructing a finite deterministic automaton from a RE was presented by Asperti et al in [5]. 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 the positions inside the RE which have been reached after reading some prefix of the input string, or better positions where the processing of the 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 appealingsness of Brzozowski's derivatives with the effectiveness of McNaughton and Yamada's labelling technique, essentially combining the best of the two 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.

The concept of prioritized transducers to formalize capturing groups in RE matching was introduced by Berglund and Merwe [36]. Their main goal was to provide an automata-based theoretical foundation for the basic functionality of modern RE matchers (with a focus on the Java RE standard library). Many RE matching libraries perform matching as a form of parsing by using capturing groups, and thus output what subexpression matched which substring. Their approach permits an analysis of matching semantics of a subset of the 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.

8. Conclusion

In this work, we presented a big-step operational semantics for a virtual machine for RE parsing. Our semantics produces, as parsing evidence, bit-codes which can be used to characterize which disambiguation strategy is followed by the semantics. In order to avoid the well-known problems with problematic REs, we use an algorithm that converts a problematic RE into an equivalent non-problematic one. All theoretical results reported in this paper are intregrally 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 a optimized Haskell library for RE parsing. The complete development is avaliable at [18].

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. Other line of research we intend to pursue is to formalize that the proposed semantics follow a disambiguation strategy and to investigate the time complexity of our algorithm.

Acknowledgements

This work is supported by the CNPq Brazil under grant No.: 426232/2016.

References

[1] A. Frisch, L. Cardelli, Greedy Regular Expression Matching, ICALP 2004 - International Colloquium on Automata, Languages and Programming 3142 (2004) 618–629.

[2] D. Firsov, T. Uustalu, 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, 2013, pp. 98–113. doi:10.1007/978-3-319-03545-1_7.
URL http://dx.doi.org/10.1007/978-3-319-03545-1_7

[3] R. Ribeiro, A. D. Bois, Certified Bit-Coded Regular Expression Parsing, Proceedings of the 21st Brazilian Symposium on Programming Languages - SBLP 2017 (2017) 1–8.
URL http://dl.acm.org/citation.cfm?doid=3125374.3125381

[4] R. Lopes, R. Ribeiro, C. Camarão, Certified derivative-based parsing of regular expressions, in: Programming Languages — Lecture Notes in Computer Science 9889, Springer, 2016, pp. 95–109.

[5] A. Asperti, C. S. Coen, E. Tassi, Regular expressions, au point, CoRR abs/1010.2604. arXiv:1010.2604.
URL http://arxiv.org/abs/1010.2604

[6] R. F. P. Lopes, Certified Derivative-based Parsing of Regular Expressions — Master Thesis.

[7] S. Fischer, F. Huch, T. Wilke, A play on regular expressions, ACM SIGPLAN Notices 45 (9) (2010) 357. doi:10.1145/1932681.1863594.
URL papers2://publication/uuid/CE64D2A8-5A9F-444C-A677-EE49DFF386D2{%}5Cnhttp://portal.acm.org/citation.cfm?doid=1932681.1863594

[8] D. E. Knuth, Top-down syntax analysis, Acta Inf. 1 (2) (1971) 79–110. doi:10.1007/BF00289517.
URL http://dx.doi.org/10.1007/BF00289517

[9] R. Cox, Regular Expression Matching: the Virtual Machine Approach.
URL https://swtch.com/{~}rsc/regexp/regexp2.html

[10] R. Ierusalimschy, A text patternmatching tool based on parsing expression grammars, Software - Practice and Experiencedoi:10.1002/spe.892.

[11] B. 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, ACM, New York, NY, USA, 2004, pp. 111–122. doi:10.1145/964001.964011.
URL http://doi.acm.org/10.1145/964001.964011

[12] M. Sulzmann, K. Z. M. Lu, Posix regular expression parsing with derivatives, in: M. Codish, E. Sumii (Eds.), Functional and Logic Programming, Springer International Publishing, Cham, 2014, pp. 203–220.

[13] L. Nielsen, F. Henglein, Bit-coded regular expression parsing, in: A.-H. Dediu, S. Inenaga, C. Martín-Vide (Eds.), Language and Automata Theory and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 402–413.

[14] J.-L. Krivine, A call-by-name lambda-calculus machine, Higher Order Symbol. Comput. 20 (3) (2007) 199–207. doi:10.1007/s10990-007-9018-9.
URL http://dx.doi.org/10.1007/s10990-007-9018-9

[15] P. J. Landin, The mechanical evaluation of expressions, The Computer Journal 6 (4) (1964) 308–320. arXiv:/oup/backfile/content_public/journal/comjnl/6/4/10.1093/comjnl/6.4.308/2/6-4-308.pdf, doi:10.1093/comjnl/6.4.308.
URL http://dx.doi.org/10.1093/comjnl/6.4.308

[16] S. Medeiros, F. Mascarenhas, R. Ierusalimschy, From regexes to parsing expression grammars, Sci. Comput. Program. 93 (2014) 3–18. doi:10.1016/j.scico.2012.11.006.
URL http://dx.doi.org/10.1016/j.scico.2012.11.006

[17] T. A. Delfino, R. Ribeiro, Towards certified virtual machine-based regular expression parsing, in: Proceedings of the XXII Brazilian Symposium on Programming Languages, SBLP '18, ACM, New York, NY, USA, 2018, pp. 67–74. doi:10.1145/3264637.3264646.
URL http://doi.acm.org/10.1145/3264637.3264646

[18] T. Delfino, R. Ribeiro, Towards certified virtual machine-based regular expression parsing — on-line repository, https://github.com/thalesad/regexvm (2018).

[19] Y. Bertot, P. Castran, Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions, 1st Edition, Springer Publishing Company, Incorporated, 2010.

[20] M. H. Sørensen, P. 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.

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

[22] A. Rathnayake, H. Thielecke, Regular Expression Matching and Operational Semantics, Electronic Proceedings in Theoretical Computer Science 62 (Sos) (2011) 31–45. arXiv:1108.3126, doi:10.4204/EPTCS.62.3.
URL http://arxiv.org/abs/1108.3126

[23] S. P. Jones (Ed.), Haskell 98 Language and Libraries: The Revised Report, http://haskell.org/, 2002.
URL http://haskell.org/definition/haskell98-report.pdf

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

[25] C. Mcbride, R. Paterson, Applicative programming with effects, J. Funct. Program. 18 (1) (2008) 1–13. doi:10.1017/S0956796807006326.
URL http://dx.doi.org/10.1017/S0956796807006326

[26] J. Kirrage, A. Rathnayake, H. Thielecke, Static analysis for regular expression denial-of-service attacks, in: J. Lopez, X. Huang, R. Sandhu (Eds.), Network and System Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 135–148.

[27] K. Claessen, J. 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, ACM, New York, NY, USA, 2000, pp. 268–279.

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

[29] S. Medeiros, R. Ierusalimschy, A parsing machine for PEGs, Proceedings of the 2008 symposium on Dynamic languages - DLS '08 (2008) 1–12doi:10.1145/1408681.1408683.
URL http://portal.acm.org/citation.cfm?doid=1408681.1408683

[30] D. Dubé, M. Feeley, Efficiently building a parse tree from a regular expression, Acta Inf. 37 (2) (2000) 121–144. doi:10.1007/s002360000037.
URL http://dx.doi.org/10.1007/s002360000037

[31] F. Ausaf, R. Dyckhoff, C. Urban, Posix lexing with derivatives of regular expressions (proof pearl), in: J. C. Blanchette, S. Merz (Eds.), Interactive Theorem Proving, Springer International Publishing, Cham, 2016, pp. 69–86.

[32] G. 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, ACM, New York, NY, USA, 2019, pp. 35–46. doi:10.1145/3294032.3294082.
URL http://doi.acm.org/10.1145/3294032.3294082

[33] G. Radanne, P. Thiemann, Regenerate: A Language Generator for Extended Regular Expressions, working paper or preprint (May 2018).
URL https://hal.archives-ouvertes.fr/hal-01788827

[34] B. Groz, S. Maneth, Efficient testing and matching of deterministic regular expressions, Journal of Computer and System Sciences 89 (2017) 372–399. doi:10.1016/j.jcss.2017.05.013.
URL http://dx.doi.org/10.1016/j.jcss.2017.05.013

[35] C. Doczkal, J. O. Kaiser, G. 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 (2013) 82–97. doi:10.1007/978-3-319-03545-1_6.

[36] M. Berglund, B. V. D. Merwe, On the semantics of regular expression parsing in the wild, Theoretical Computer Science 1 (2016) 1–14. doi:10.1016/j.tcs.2016.09.006.
URL http://dx.doi.org/10.1016/j.tcs.2016.09.006

***Potential Reviewers**

Sérgio Medeiros – sergiomedeiros@ect.ufrn.br
Roberto Ierusalimsky – roberto@inf.puc-rio.br
Martin Sulzmann – martin.sulzmann@gmail.com