The Design of a Verified Derivative-Based Parsing Tool for Regular Expressions

Elton M. Cardoso ; Maycon J. Amaro
Universidade Federal de Ouro Preto, Departamento de Computação,
Ouro Preto - MG, Brazil
eltonmc@ufop.edu.br ; maycon.amaro@aluno.ufop.edu.br

and

Samuel da Silva Feitosa
Instituto Federal de Santa Catarina
Caçador - SC, Brazil
samuel.feitosa@ifsc.edu.br

and

Leonardo V. S. Reis
Universidade Federal de Juiz de Fora
Juiz de Fora - MG, Brazil
lvsreis@ice.ufjf.br

and

Andre Rauber Du Bois
Universidade Federal Pelotas
Pelotas - RS, Brazil
dubois@inf.ufpel.edu.br

and

Rodrigo G. Ribeiro
Universidade Federal de Ouro Preto, Departamento de Computação,
Ouro Preto - MG, Brazil
rodrigo.ribeiro@ufop.edu.br

Abstract

We describe the formalization of Brzozowski and Antimirov derivative based algorithms for regular expression parsing, in the dependently typed language Agda. The formalization produces a proof that either an input string matches a given regular expression or that no matching exists. A tool for regular expression based search in the style of the well known GNU grep has been developed with the certified algorithms. Practical experiments conducted with this tool are reported.

Keywords: Certified algorithms, regular expressions, dependent types.

1 Introduction

Parsing is the process of analysing if a string of symbols conforms to a given set of rules. In computer science, it involves the formal specification of the rules in a grammar. An important product of parsing is construction of data that makes evident which rules have been used to conclude that the string of symbols can be obtained from the grammar rules or the indication of an error that represents the fact that the string of symbols cannot be generated from the grammar rules.

In this work we are interested in the parsing problem for regular languages (RLs) [1], i.e. languages that can be recognized by (non-) deterministic finite automata and equivalent formalisms. Regular expressions (REs) are an algebraic and compact way of specifying RLs that are extensively used in lexical analyser generators [2] and string search utilities [3]. Since such tools are widely used and parsing is pervasive in computing, there is a growing interest on certified parsing algorithms [4–6]. This interest is motivated by the recent development of dependently typed languages which are powerful enough to express algorithmic properties as types, that are automatically checked by a compiler.

The use of derivatives for regular expressions were introduced by Brzozowski [7] as an alternative method to compute a finite state machine that is equivalent to a given RE and to perform RE-based parsing. According to Owens et. al [8], "derivatives have been lost in the sands of time" until their work on functional encoding of RE derivatives have renewed interest on their use for parsing [9, 10]. In this work, we provide a complete formalization of an algorithm for RE parsing using derivatives [8], and describe a RE based search tool we developed by using the dependently typed language Agda [11]. We want to emphasize that what we call "RE parsing" is the problem of finding all prefixes and substrings of an input that matches a given RE, as in RE based text search tools as GNU-grep [3].

More specifically, our contributions are:

- A formalization of Brzozowski derivatives based RE parsing in Agda. The presented certified algorithm produces as a result either a proof term (parse tree) that is evidence that the input string is in the language of the input RE or a witness that such parse tree does not exist.
- A detailed explanation of the technique used to simplify derivatives using "smart-constructors" [8]. We give formal proofs that smart constructors indeed preserve the language recognized by REs.
- A formalization of Antimirov's partial derivatives and their use to construct a RE parsing algorithm. The main difference between partial derivatives and Brzozowski's is that the former computes a set of REs using set operators instead of "smart-constructors". Producing a set of REs avoids the need of simplification using smart constructors.
- We use the verified algorithms to build a certified RE matching tool, in the style of well-known GNU-grep [3], and use it in some experiments.

The rest of this paper is organized as follows. Section 2 presents a brief introduction to Agda. Section 3 describes the encoding of REs and its parse trees. In Section 4 we define Brzozowski and Antimirov derivatives and smart constructors and some of their properties, describing how to build a correct parsing algorithm from them. Section 5 comments on the use of the certified algorithm to build a tool for RE-based search and present some experiments with this tool. Related work is discussed on Section 6 and Section 7 concludes our work.

All the source code in this article has been formalized in Agda version 2.5.2 using Standard Library 0.13, but we do not present every detail. Proofs of some properties result in functions with a long pattern matching structure, that would distract the reader from understanding the high-level structure of the formalization. In such situations we give just proof sketches. All details can be found in the source code available at [12].

2 An Overview of Agda

Agda is a dependently-typed functional programming language based on Martin-Löf intuitionistic type theory [13]. Function types and an infinite hierarchy of types, $\mathsf{Set}\ l$, where l is a natural number, are built-in. Everything else is a user-defined type. The type Set, also known as Set_0, is the type of all "small" types, such as Bool, String and $\mathsf{List}\ \mathsf{Bool}$. The type Set_1 is the type of Set and "others like it", such as $\mathsf{Set} \to \mathsf{Bool}$, $\mathsf{String} \to \mathsf{Set}$, and $\mathsf{Set} \to \mathsf{Set}$. We have that $\mathsf{Set}\ l$ is an element of the type $\mathsf{Set}\ (l + 1)$, for every $l \geqslant 0$. This stratification of types is used to keep Agda consistent as a logical theory [14].

An ordinary (non-dependent) function type is written $A \to B$ and a dependent one is written $(x : A) \to B$, where type B depends on x, or $\forall\ (x : A) \to B$. Agda allows the definition of *implicit parameters*, i.e. parameters whose value can be infered from the context, by surrounding them in curly

braces: $\forall\,\{x : A\} \to B$. To avoid clutter, we'll omit implicit arguments from the source code presentation. The reader can safely assume that every free variable in a type is an implicity parameter.

As an example of Agda code, consider the the following data type of length-indexed lists, also known as vectors.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)
```

The Vec type uses some interesting concepts. First, Vec is *parameterized* by a type A : Set, which means that in every occurrence of Vec its parameter A should not change. Second, the type of Vec A is $\mathbb{N} \to$ Set, i.e. Vec A is a family of types indexed by natural numbers. For each natural number n, Vec A n is a type.

In Vec's definition, constructor [] builds empty vectors. The cons-operator (_ :: _) inserts a new element in front of a vector of n elements (of type Vec A n) and returns a value of type Vec A (suc n). The Vec datatype is an example of a dependent type, i.e. a type that uses a value (that denotes its length). The usefulness of dependent types can be illustrated with the definition of a safe list head function: head can be defined to accept only non-empty vectors, i.e. values of type Vec A (suc n).

```
head : Vec A (suc n) → A
head (x :: xs) = x
```

In head's definition, constructor [] is not used. The Agda type-checker can figure out, from head's parameter type, that argument [] to head is not type-correct.

Thanks to the propositions-as-types principle[1] we can interpret types as logical formulas and terms as proofs. We can encode logical disjunction as the following Agda type:

```
data _∨_ (A B : Set) : Set where
  inj₁ : A → A ∨ B
  inj₂ : B → A ∨ B
```

Note that each constructor of the previous data type represents how we can build evidence for the proposition $A \vee B$: using constructor inj_1 together with an evidence for A or using constructor inj_2 and evidence for B. Intuitively, type _∨_ encodes the intuitionistic intepretation of disjunction. Another important example is the representation of equality as the following Agda type:

```
data _≡_ {l} {A : Set l} (x : A) : A → Set where
  refl : x ≡ x
```

This type is called propositional equality. It defines that there is a unique evidence for equality, constructor refl (for reflexivity), that asserts that the only value equal to x is itself. Given a type P, type Dec P is used to build proofs that P is a decidable proposition, i.e. that either P or not P holds. The decidable proposition type is defined as:

```
data Dec (P : Set) : Set where
  yes : P → Dec P
  no : ¬ P → Dec P
```

Constructor yes stores a proof that property P holds and constructor no an evidence that such proof is impossible. Some functions used in our formalization use this type. The type $\neg\ P$ is an abbreviation for $P \to \bot$, where $\bot$ is a data type with no constructors (i.e. a data type for which it is not possible to construct a value, which corresponds to a false proposition).

Dependently typed pattern matching is built by using the so-called **with** construct, that allows for matching intermediate values [15]. If the matched value has a dependent type, then its result can affect the form of other values. For example, consider the following function that tests the equality of two $\mathbb{N}$ values.

```
sucInv : (suc n) ≡ (suc m) → n = m
sucInv refl = refl
```

[1] Also known as Curry-Howard "isomorphism" [14].

```
_ ≟ _ : (n m : ℕ) → Dec (n ≡ m)
zero ≟ zero = yes refl
zero ≟ suc m = no (λ ())
suc n ≟ zero = no (λ ())
suc n ≟ suc m with n ≟ m
(suc n) ≟ (suc .n) | yes refl = yes refl
(suc n) ≟ (suc m) | no p = no (p ∘ sucInv)
```

Notice that when we pattern-match on the result of the recursive call $n \stackrel{?}{=} m$ using the **with** construct, the value of the second parameter is specialized using the fact that yes refl denotes that both n and m must be equal. For the case that n and m are different, we use the inversion lemma sucInv, whose meaning is immediate, to derive a contradiction. For further information about Agda, see [11,16].

3 Regular Expressions

Regular expressions are defined with respect to a given alphabet. Formally, RE syntax is defined by the following context-free grammar

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

where a is any symbol from the underlying alphabet. In our formalization, we represent alphabet symbols using Agda's type Char.

Datatype Regex encodes RE syntax.

```
data Regex : Set where
  ∅ : Regex
  ϵ : Regex
  $_ : Char → Regex
  _•_ : Regex → Regex → Regex
  _+_ : Regex → Regex → Regex
  _⋆ : Regex → Regex
```

Constructors $\emptyset$ and ϵ denote respectively the empty language ($\emptyset$) and the empty string (ϵ). Alphabet symbols are constructed by using the \$ constructor. Composite REs are built using concatenation ($\bullet$), union ($+$) and Kleene star ($\star$).

We define RE semantics as the inductively defined judgment $s \in e$, which means that string s is in the language denoted by RE e.

$$\frac{}{\epsilon \in \epsilon}\ Eps \qquad \frac{}{a \in a}\ Chr \qquad \frac{s \in e \quad s' \in e'}{ss' \in ee'}\ Cat$$

$$\frac{s \in e}{s \in e + e'}\ Left \qquad \frac{s \in e'}{s \in e + e'}\ Right \qquad \frac{}{\epsilon \in e^\star}\ StarBase$$

$$\frac{s \in e \quad s' \in e^\star}{ss' \in e^\star}\ StarRec$$

Rule *Eps* specifies that only the empty string is accepted by RE ϵ, while rule *Chr* says that a single character string is accepted by the RE formed by this character. The rules for concatenation and choice are straightforward. For Kleene star, we need two rules: the first specifies that the empty string is in the language of RE $e^\star$ and rule *StarRec* says that the string ss' is in the language denoted by $e^\star$ if $s \in e$ and $s' \in e^\star$.

In our formalization, we represent strings as values of type List Char and we encode the RE semantics as an inductive data type in which each constructor represents a rule for the previously defined semantics. Agda allows the overloading of constructor names. In some cases we use the same symbol both in the RE syntax and in the definition of its semantics.

```
data _ ∈ ⟦_⟧ : List Char → Regex → Set where
  ϵ : [] ∈ ⟦ ϵ ⟧
  $_ : (a : Char) → [ a ] ∈ ⟦ $ a ⟧
```

```
_•_⇒_ : xs ∈ ⟦ l ⟧ → ys ∈ ⟦ r ⟧ →
  zs ≡ xs ++ ys → zs ∈ ⟦ l • r ⟧
_+L_ : (r : Regex) → xs ∈ ⟦ l ⟧ → xs ∈ ⟦ l + r ⟧
_+R_ : (l : Regex) → xs ∈ ⟦ r ⟧ → xs ∈ ⟦ l + r ⟧
_⋆ : xs ∈ ⟦ ϵ + (e • e ⋆) ⟧ → xs ∈ ⟦ e ⋆ ⟧
```

Constructor ϵ states that the empty string (denoted by the empty list $[\,]$) is in the language of RE ϵ.

For any single character a, the singleton string $[\ a\ ]$ is in the RL for $\$\ a$. Given parse trees for REs l and r, $xs \in \llbracket\ l\ \rrbracket$ and $ys \in \llbracket\ r\ \rrbracket$, constructor $\_ \bullet \_ \Rightarrow \_$ can be used to build a parse tree for the concatenation of these REs. Constructor $\_ + \mathsf{L}\_$ ($\_ + \mathsf{R}\_$) creates a parse tree for $l + r$ from a parse tree for l (r). Parse trees for Kleene star are built using the following well known equivalence of REs: $e^\star = \epsilon + e\,e^\star$.

Several inversion lemmas about RE parsing relation are necessary for derivative-based parsing formalization. They consist of pattern-matching on proofs of $\_ \in \llbracket\_\rrbracket$. As an example, we present below the inversion lemma for a single character RE.

```
charInvert : x :: xs ∈ ⟦ $ y ⟧ → x ≡ y × xs ≡ [ ]
charInvert ($ c) = refl , refl
```

Intuitively, function charInvert specifies that if a string $x :: xs$ matches RE $\$\ y$ then the input string must be a single character string, i.e. $xs \equiv [\,]$ and $x \equiv y$. Other inversions lemmas are defined in our formalization. They follow the same structure of charInvert: producing, as result, the necessary conditions for a RE semantics proof to hold and are omitted for brevity.

4 Derivatives, Smart Constructors and Parsing

Formally, the derivative of a formal language $L \subseteq \Sigma^\star$ with respect to a symbol $a \in \Sigma$ is the language formed by suffixes of L words without the prefix a.

An algorithm for computing the derivative of a language represented by a RE as another RE is due to Brzozowski [7]. It relies on a function (called ν) that determines if some RE accepts or not the empty string (by returning ϵ or $\emptyset$, respectively):

$$
\begin{array}{lcl}
\nu(\emptyset) & = & \emptyset \\
\nu(\epsilon) & = & \epsilon \\
\nu(a) & = & \emptyset \\
\nu(e\,e') & = & \begin{cases} \epsilon & \text{if } \nu(e) = \nu(e') = \epsilon \\ \emptyset & \text{otherwise} \end{cases} \\
\nu(e + e') & = & \begin{cases} \epsilon & \text{if } \nu(e) = \epsilon \text{ or } \nu(e') = \epsilon \\ \emptyset & \text{otherwise} \end{cases} \\
\nu(e^\star) & = & \epsilon
\end{array}
$$

Decidability of $\nu(e)$ is proved by function $\nu[\_]$, which is defined by induction over the structure of the input RE e and returns a proof that the empty string is accepted or not, using Agda type of decidable propositions, Dec P.

```
ν[_] : ∀ (e : Regex) → Dec ([ ] ∈ ⟦ e ⟧)
ν[ ∅ ] = no (λ ())
ν[ ϵ ] = yes ϵ
ν[ $ x ] = no (λ ())
ν[ e • e' ] with ν[ e ] | ν[ e' ]
...| yes pr | (yes pr1) = yes (pr • pr1 ⇒ refl)
...| yes pr | (no ¬pr1) = no (¬pr1 ∘ π₂ ∘ •invert)
...| no ¬pr | pr1 = no (¬pr ∘ π₁ ∘ •invert)
ν[ e + e' ] with ν[ e ] | ν[ e' ]
...| yes pr | pr1 = yes (e' +L pr)
...| no ¬pr | (yes pr1) = yes (e +R pr1)
...| no ¬pr | (no ¬pr1)
   = no ([ ¬pr , ¬pr1 ] ∘ +invert)
ν[ e ⋆ ] = yes ((e • e ⋆ +L ϵ) ⋆)
```

The definition of $\nu[\_]$ uses some of the inversion lemmas about RE semantics. Lemma •invert states that if the empty string is in the language of $l \bullet r$ (where l and r are arbitrary REs) then the empty string belongs to l and r's languages. Lemma +invert is defined similarly for union.

4.1 Smart Constructors

In order to define Brzozowski derivatives, we follow Owens et. al. [8]. We use smart constructors to identify equivalent REs modulo identity and nullable elements, ϵ and $\emptyset$, respectively. RE equivalence is denoted by $e \approx e'$ and it's defined as usual [1]. The equivalence axioms maintained by smart constructors are:

$$\begin{array}{llllllll} 1) & e + \emptyset \approx e & 2) & \emptyset + e \approx e & 3) & e\,\emptyset \approx \emptyset & 4) & e\,\epsilon \approx e \\ 5) & \emptyset\, e \approx \emptyset & 6) & \epsilon\, e \approx e & 7) & \emptyset^\star \approx \epsilon & 8) & \epsilon^\star \approx \epsilon \end{array}$$

These axioms are kept as invariants using functions that preserve them while building REs. As a convention, a smart constructor is named by prefixing the constructor name with a back quote. In the case of union, the definition of the smart constructor differs only when one the parameters denotes the empty language:

```
_`+_ : (e e′ : Regex) → Regex
∅ `+ e′ = e′
e `+ ∅ = e
e `+ e′ = e + e′
```

In the case of concatenation, we need to deal with the possibilities of each parameter being empty (denoting the empty language) or the empty string. If one of them is empty ($\emptyset$) the result is also empty, and the empty string is the identity for concatenation.

```
_`•_ : (e e′ : Regex) → Regex
∅ `• e′ = ∅
ϵ `• e′ = e′
e `• ∅ = ∅
e `• ϵ = e
e `• e′ = e • e′
```

For Kleene star both $\emptyset$ and ϵ are replaced by ϵ.

```
_`⋆ : Regex → Regex
∅ `⋆ = ϵ
ϵ `⋆ = ϵ
e `⋆ = e ⋆
```

Since all smart constructors produce equivalent REs, they preserve the parsing relation. This property is stated below as a soundness and completeness lemma of each smart constructor with respect to RE membership proofs.

Lemma 1 (Soundness and completeness of union)**.** *For all REs e, e' and all strings xs, $xs \in [\![\ e\ `+\ e'\ ]\!]$ holds if and only if, $xs \in [\![\ e + e'\ ]\!]$ also holds.*

Lemma 2 (Soundness and completeness of concatenation)**.** *For all REs e, e' and all strings xs, $xs \in [\![\ e\ `\bullet\ e'\ ]\!]$ holds if and only if, $xs \in [\![\ e \bullet e'\ ]\!]$ also holds.*

Lemma 3 (Soundness and completeness of Kleene star)**.** *For all REs e and string xs, $xs \in [\![\ e\ `\star\ ]\!]$ holds if and only if, $xs \in [\![\ e \star\ ]\!]$ also holds.*

4.2 Brzozowski Derivatives and their Properties

Intuitively, the derivative L_a of a language L with respect to a symbol a is the set of strings generated by stripping the leading a from the strings in L that start with a. Formally: $L_a = \{w \mid aw \in L\}$. Regular languages are closed under the derivative operation and Janusz Brzozowski defined a elegant method for computing derivatives for RE in [7]. Formally, the derivative of a RE e with respect to a symbol a, denoted by $\partial_a(e)$, is defined by recursion on e's structure as follows:

$$\begin{array}{lcl} \partial_a(\emptyset) & = & \emptyset \\ \partial_a(\epsilon) & = & \emptyset \\ \partial_a(b) & = & \begin{cases} \epsilon & \text{if } b = a \\ \emptyset & \text{otherwise} \end{cases} \\ \partial_a(e\,e') & = & \partial_a(e)\,e' + \nu(e)\,\partial_a(e') \\ \partial_a(e + e') & = & \partial_a(e) + \partial_a(e') \\ \partial_a(e^\star) & = & \partial_a(e)\,e^\star \end{array}$$

This function has an immediate translation to Agda. Notice that the derivative function uses smart constructors to quotient result REs with respect to the equivalence axioms presented in Section 4.1 and RE emptiness test. In the symbol case (constructor $\$\_$), function $\stackrel{?}{=}$ is used to produce an evidence for equality of two Char values.

```
∂[_,_] : Regex → Char → Regex
∂[ ∅ , c ] = ∅
∂[ ϵ , c ] = ∅
∂[ $ c , c′ ] with c ≟ c′
...| yes refl = ϵ
...| no prf = ∅
∂[ e • e′ , c ] with ν[ e ]
... | yes pr = (∂[ e , c ] ‘• e′) ‘+ ∂[ e′ , c ]
... | no ¬pr = ∂[ e , c ] ‘• e′
∂[ e + e′ , c ] = ∂[ e , c ] ‘+ ∂[ e′ , c ]
∂[ e ⋆ , c ] = ∂[ e , c ] ‘• (e ‘⋆)
```

From this definition we prove the following important properties of the derivative operation. Soundness of $\partial[\_,\_]$ ensures that if a string xs is in the language of $\partial[\ e\ ,\ x\ ]$, then $(x :: xs) \in [\![\ e\ ]\!]$ holds. Completeness ensures that the other direction of implication holds. Both are proved by induction on e's structure using previously defined lemmas.

Theorem 1 (Derivative operation soundness and completeness)**.** *For all REs e, all strings xs and all symbols x, $xs \in [\![\ \partial[\ e\ ,\ x\ ]\ ]\!]$ holds if, and only if, $(x :: xs) \in [\![\ e\ ]\!]$ holds.*

Proof. Both directions are proved by induction on the structure of e, using the soundness (completeness) lemmas for smart constructors and decidability of the emptiness test. □

Theorem 2 (Derivative operation completeness)**.** *For all REs e, all strings xs and all symbols x, if $(x :: xs) \in [\![\ e\ ]\!]$ holds then $xs \in [\![\ \partial[\ e\ ,\ x\ ]\ ]\!]$ holds.*

Proof. By induction on the structure of e using the completeness lemmas for smart constructors and decidability of the emptiness test. □

4.3 Antimirov's Partial Derivatives and its Properties

RE derivatives were introduced by Brzozowski to construct a DFA (deterministic finite automata) from a given RE. Partial derivatives were introduced by Antimirov [17] as a method to construct a NFA (non-deterministic finite automata). The main insight of partial derivatives for building NFAs is building a set of REs which collectively accept the same strings as Brzozowski derivatives. Algebraic properties of set operations ensures that ACUI[2] equations hold. Below, we present function $\nabla_a(e)$ which computes the set of partial derivatives from a given RE e and a symbol a.

$$
\begin{array}{lcl}
\nabla_a(\emptyset) & = & \emptyset \\
\nabla_a(\epsilon) & = & \emptyset \\
\nabla_a(b) & = & \begin{cases} \{\epsilon\} & \text{if } b = a \\ \emptyset & \text{otherwise} \end{cases} \\
\nabla_a(e\ e') & = & \begin{cases} \nabla_a(e) \odot e' \cup \nabla_a(e') & \text{if } \nu(e) = \epsilon \\ \nabla_a(e) \odot e' & \text{otherwise} \end{cases} \\
\nabla_a(e + e') & = & \nabla_a(e) \cup \nabla_a(e') \\
\nabla_a(e^\star) & = & \nabla_a(e) \odot e^\star
\end{array}
$$

Function $\nabla_a(e)$ uses the operator $S \odot e'$ which concatenates RE e' at the right of every $e \in S$, i.e. $S \odot e' = \{e \bullet e' \mid e \in S\}$.

Our Agda implementation models sets as lists of regular expressions.

```
Regexes = List Regex
```

The operator that concatenates a RE at the right of every $e \in S$ is defined by induction on S:

[2] Associativity, Commutativity and Idempotence with Unit elements axioms for REs [7].

```
_⊙_ : Regexes → Regex → Regexes
[] ⊙ e = []
(e′ :: es′) ⊙ e = (e′ • e) :: (es′ ⊙ e)
```

The definition of a function to compute partial derivatives for a given RE is a direct translation of mathematical notation to Agda code:

```
∇[_,_] : Regex → Char → Regexes
∇[ ∅ , c ] = []
∇[ ϵ , c ] = []
∇[ $ x , c ] with x ≟ c
... | yes refl = [ ϵ ]
... | no p = []
∇[ e • e′ , c ] with ν[ e ]
... | yes p = (e′ ⊙ ∇[ e , c ]) ++ ∇[ e′ , c ]
... | no ¬p = e′ ⊙ ∇[ e , c ]
∇[ e + e′ , c ] = ∇[ e , c ] ++ ∇[ e′ , c ]
∇[ e ⋆ , c ] = (e ⋆) ⊙ ∇[ e , c ]
```

In order to prove relevant properties about partial derivatives, we define a relation that specifies when a string is accepted by some set of REs.

```
data _∈⟨⟨_⟩⟩ : List Char → Regexes → Set where
  here : s ∈ ⟦ e ⟧ → s ∈⟨⟨ e :: es ⟩⟩
  there : s ∈⟨⟨ es ⟩⟩ → s ∈⟨⟨ e :: es ⟩⟩
```

Essentially, a value of type $s \in \langle\langle\ S\ \rangle\rangle$ indicates that s is accepted by some RE in S. The next lemmas on the membership relation $s \in \langle\langle\ S\ \rangle\rangle$ and list concatenation are used to prove soundness and completeness of partial derivatives.

Lemma 4 (Weakening left)**.** *For all sets of REs S, S' and all strings s, if $s \in \langle\langle\ S\ \rangle\rangle$ holds then $s \in \langle\langle\ S +\!\!+ S'\ \rangle\rangle$ holds.*

Lemma 5 (Weakening right)**.** *For all sets of REs S, S' and all strings s, if $s \in \langle\langle\ S'\ \rangle\rangle$ holds then $s \in \langle\langle\ S +\!\!+ S'\ \rangle\rangle$ holds.*

Lemma 6. *For all sets of REs S, S' and all strings s, if $s \in \langle\langle\ S +\!\!+ S'\ \rangle\rangle$ holds then $s \in \langle\langle\ S\ \rangle\rangle$ or $s \in \langle\langle\ S'\ \rangle\rangle$ holds.*

Lemma 7. *For all sets of REs S, all REs e and all strings s, s', if $s \in \langle\langle\ S\ \rangle\rangle$ and $s' \in [\![\ e\ ]\!]$ holds then $s +\!\!+ s' \in \langle\langle\ e \odot S\ \rangle\rangle$ holds.*

Lemma 8. *For all sets of REs S, all REs e and all strings s, if $s \in \langle\langle\ e \odot S\ \rangle\rangle$ holds then there exist s_1 and s_2 such that $s \equiv s_1 +\!\!+ s_2$, $s_1 \in \langle\langle\ S\ \rangle\rangle$ and $s_2 \in [\![\ e\ ]\!]$ holds.*

Using these previous results about $\_ \in \langle\langle\_\rangle\rangle$, we can enunciate soundness and completeness theorems of partial derivatives. Let e be an arbitrary RE and a an arbitrary symbol. Soundness means that if a string s is accepted by some RE in $\nabla[\ e\ ,\ a\ ]$ then $(a :: s) \in [\![\ e\ ]\!]$. The completeness theorem shows that the other direction of the soundness implication also holds. Both implications hold by induction on e's structure.

Theorem 3 (Partial derivative operation soundness and completeness)**.** *For all symbols a, all strings s and all REs e, we have that $s \in \langle\langle\ \nabla[\ e\ ,\ a\ ]\ \rangle\rangle$ holds, if and only if, $(a :: s) \in [\![\ e\ ]\!]$ holds.*

Proof. Both directions hold by induction on the structure of e, using Lemmas 4, 5, 6, 7 and 8. □

Theorem 4 (Partial derivative operation completeness)**.** *For all symbols a, all strings s and all REs e, if $(a :: s) \in [\![\ e\ ]\!]$ holds then $s \in \langle\langle\ \nabla[\ e\ ,\ a\ ]\ \rangle\rangle$ holds.*

Proof. Induction on the structure of e, using Lemmas 4, 5 and 7. □

4.4 Parsing

Assume that we are given an RE e and a string s and we want to determine if $s \in e$. We can use RE derivatives for building such a test by extending the definition of derivatives to strings as follows [8]:

$$\begin{array}{lcl} \widehat{\partial}_{\epsilon}(e) & = & e \\ \widehat{\partial}_{as}(e) & = & \widehat{\partial}_{s}(\partial_{a}(e)) \end{array}$$

Note that $s \in e$ if and only if $\epsilon \in \widehat{\partial}_s(e)$, which is true whenever $\nu(\widehat{\partial}_s(e)) = \epsilon$. Owens et. al. define a relation between strings and RE, called the *matching* relation, defined as:

$$\begin{array}{lcl} e \sim \epsilon & \Leftrightarrow & \nu(e) \\ e \sim as & \Leftrightarrow & \partial_a(e) \sim s \end{array}$$

A simple inductive proof shows that $s \in e$ if and only if $e \sim s$.

For our purposes, understanding RE parsing as a matching relation isn't adequate because RE-based text search tools, like GNU-grep, shows every matching prefix and substring of a RE for a given input. Since our interest is in determining which prefixes and substrings of the input string match a given RE, we define datatypes that represent the fact that a given RE matches a prefix or a substring of some string.

We say that RE e matches a prefix of string xs if there exist strings ys and zs such that $xs \equiv ys \mathbin{+\!\!+} zs$ and $ys \in [\![\ e\ ]\!]$. Definition of IsPrefix datatype encodes this concept. Datatype IsSub specifies when a RE e matches a substring in xs: there must exist strings ys, zs and ws such that $xs \equiv ys \mathbin{+\!\!+} zs \mathbin{+\!\!+} ws$ and $zs \in [\![\ e\ ]\!]$ hold. We could represent prefix and substring predicates using dependent products, but for code clarity we choose to define the types IsPrefix and IsSub.

```
data IsPrefix (xs : List Char) (e : Regex) : Set where
  Prefix : xs ≡ ys ++ zs → ys ∈ [[ e ]] → IsPrefix xs e
data IsSub (xs : List Char) (e : Regex) : Set where
  Sub : xs ≡ ys ++ zs ++ ws → zs ∈ [[ e ]] → IsSub xs e
```

Using these datatypes we can state and prove the following relevant properties of prefixes and substrings which are immediate consequences of these definitions.

Lemma 9 (Lemma ¬IsPrefix)**.** *For all REs e, if $[\,] \in [\![\ e\ ]\!]$ does not hold then neither does* IsPrefix $[\,]$ e.

Proof. Immediate from the definition of IsPrefix and properties of list concatenation. □

Lemma 10 (Lemma ¬IsPrefix– ::)**.** *For all REs e and all strings xs, if $[\,] \in [\![\ e\ ]\!]$ and* IsPrefix xs $\partial[\ e\ ,\ x\ ]$ *do not hold then neither does* IsPrefix $(x :: xs)$ e.

Proof. Immediate from the definition of IsPrefix and Theorem 2. □

Lemma 11 (Lemma ¬IsSubstring)**.** *For all REs e, if* IsPrefix $[\,]$ e *does not hold then neither does* IsSub $[\,]$ e.

Proof. Immediate from the definitions of IsPrefix and IsSub. □

Lemma 12 (Lemma ¬IsSubstring– ::)**.** *For all strings xs, all symbols x and all REs e, if* IsPrefix $(x :: xs)$ e *and* IsSub xs e *do not hold then neither does* IsSub $(x :: xs)$ e.

Proof. Immediate from the definitions of IsPrefix and IsSub. □

Function IsPrefixDec decides if a given RE e matches a prefix in xs by induction on the structure of xs, using Lemmas 9, 10, decidable emptyness test $\nu[\_]$ and Theorem 1. Intuitively, IsPrefixDec first checks if current RE e accepts the empty string. In this case, $[\,]$ is returned as a prefix. Otherwise, it verifies, for each symbol x, whether RE $\partial[\ e\ ,\ x\ ]$ matches a prefix of the input string. If this is the case, a prefix including x is built from a recursive call to IsPrefixDec or if no prefix is matched a proof of such impossibility is constructed using lemma 10.

```
IsPrefixDec : ∀ xs e → Dec (IsPrefix xs e)
IsPrefixDec [] e with ν[ e ]
IsPrefixDec [] e | yes p = yes (Prefix [] [] refl p)
IsPrefixDec [] e | no ¬p = no (¬IsPrefix ¬p)
IsPrefixDec (x :: xs) e with ν[ e ]
... | yes p = yes (Prefix [] (x :: xs) refl p)
```

```
... | no ¬p with IsPrefixDec xs (∂[ e , x ])
... | no ¬p | (yes (Prefix ys zs eq wit))
    = yes (Prefix (x :: ys) zs (cong (_ :: _ x) eq)
      (∂ − sound _ _ _ wit))
... | no ¬pn | (no ¬p)
    = no (¬IsPrefix− :: ¬pn ¬p)
```

Function IsSubDec is also defined by induction on the structure of the input string xs, using IsPrefixDec to check whether it is possible to match a prefix of e. In this case, a substring is built from this prefix. If there's no such prefix, a recursive call is made to check if there is a substring match, returning such substring or a proof that it does not exist.

```
IsSubDec : ∀ xs e → Dec (IsSub xs e)
IsSubDec [ ] e with ν[ e ]
... | yes p = yes (Sub [ ] [ ] [ ] refl p)
... | no ¬p = no (¬IsSubstring (¬IsPrefix ¬p))
IsSubDec (x :: xs) e with IsPrefixDec (x :: xs) e
... | yes (Prefix ys zs eq wit)
    = yes (Sub [ ] ys zs eq wit)
... | no ¬p with IsSubDec xs e
... | no ¬p | (yes (Sub ys zs ws eq wit))
    = yes (Sub (x :: ys) zs ws (cong (_ :: _ x) eq) wit)
... | no ¬p1 | (no ¬p)
    = no (¬IsSubstring− :: ¬p1 ¬p)
```

Previously defined functions for computing prefixes and substrings use Brzozowski derivatives. Functions for building prefixes and substrings using Antimirov's partial derivatives are similar to Brzozowski derivative based ones. The main differences between them are in the necessary lemmas used to prove decidability of the prefix and substring relations. Such lemmas are slightly modified versions of Lemmas 9 and 10 that consider the relation $\_ \in \langle\langle \_ \rangle\rangle$ and are omitted for brevity.

5 Implementation Details and Experiments

From the formalized algorithm we built a tool for RE parsing in the style of GNU Grep [3]. We have built a simple parser combinator library for parsing RE syntax, using the Agda Standard Library and its support for calling Haskell functions through its foreign function interface.

Experimentation with our tool involved a comparison of its performance with GNU Grep [3] (grep), Google regular expression library re2 [18] and Haskell RE parsing algorithms haskell-regexp, described in [10]. We run RE parsing experiments on a machine with a Intel Core I7 1.7 GHz, 8GB RAM running Mac OS X 10.12.3; the results were collected and the median of several test runs was computed.

We used the same experiments as those used in [19]; these consist of parsing files containing thousands of occurrences of symbol `a`, using the RE $(a+b+ab)^\star$, and parsing files containing thousands of occurrences of `ab`, using the same RE. Results are presented in Figure 1 and 2, respectively.

When compared with highly-optimized tools like grep or Google's re2 library, our tool behaves poorly but, our implementation has a performance similar to algorithms developed by [10]. The cause of this inefficiency needs further investigation, but we envisaged that it can be due to the following: 1) Our algorithm relies on the Brzozowski's definition for RE parsing, which needs to quotient resulting REs. 2) We use lists to represent sets of Antimirov's partial derivatives. We believe that usage of better data structures to represent sets and using appropriate disambiguation strategies like greedy parsing [20] and POSIX [19] would be able to improve the efficiency of our algorithm without sacrificing correctness. We leave the formalization of disambiguation strategies and the use of more efficient data structures for future work.

6 Related Work

Recently, derivative-based parsing has received a lot of attention. Owens et al. were the first to present a functional encoding of RE derivatives and use it to parsing and DFA building. They use derivatives to build scanner generators for ML and Scheme [8]; no formal proof of correctness was presented.

Might et al. [9] report on the use of derivatives for parsing not only RLs but also context-free ones. They use derivatives to handle context-free grammars (CFG) and develops an equational theory for compaction that allows for efficient CFG parsing using derivatives. Implementation of derivatives for CFGs are described

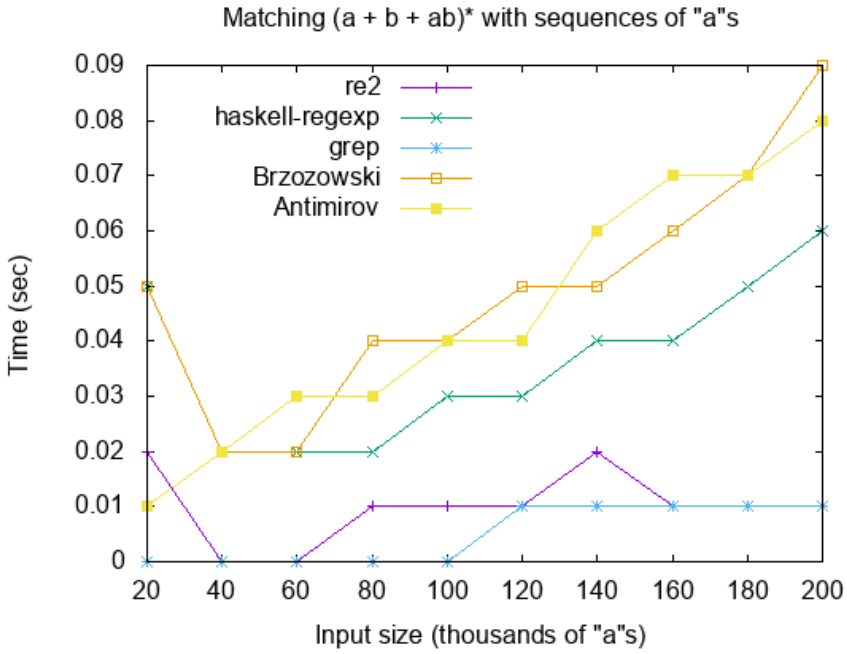

Figure 1: Results of experiment 1.

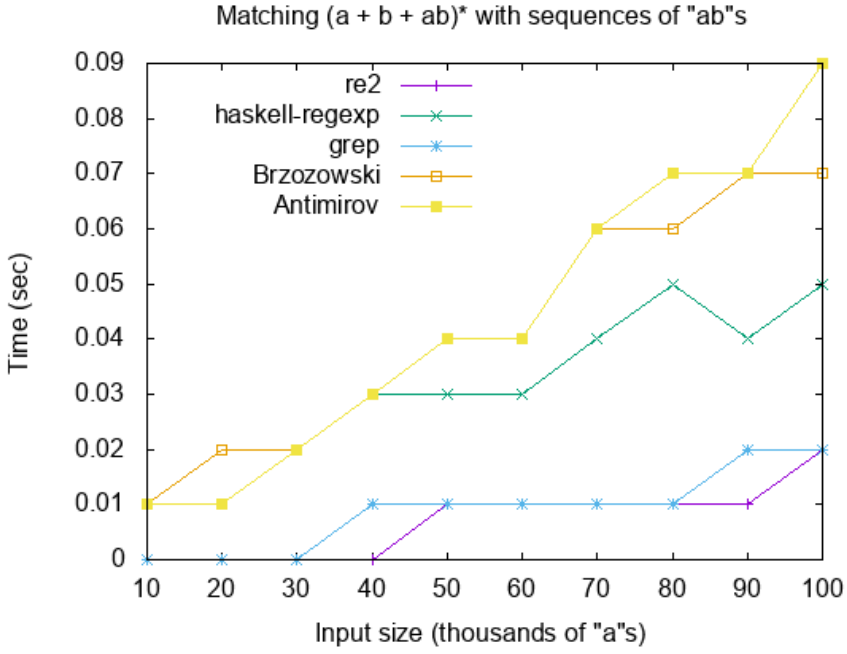

Figure 2: Results of experiment 2.

by using the Racket programming language [21]. However, Might et al. do not present formal proofs related to the use of derivatives for CFGs.

Fischer et al. describe an algorithm for RE-based parsing based on weighted automata in Haskell [10]. The paper describes the design evolution of such algorithm as a dialog between three persons. Their implementation has a competitive performance when compared with Google's RE library [18]. This work also does not consider formal proofs of RE parsing.

An algorithm for POSIX RE parsing is described in [19]. 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.

Certified algorithms for parsing also received attention recently. Firsov et al. describe a certified algorithm for RE parsing by converting an input RE to an equivalent NFA represented as a boolean matrix [4]. A matrix library based on some "block" operations [22] was developed and used to construct an Agda formalization of NFA-based parsing [11]. Compared to our work, a NFA-based formalization requires much more infrastructure (such as a Matrix library). No experiments with the certified algorithm were reported.

Almeida et al. [23] describe a Coq formalization of partial derivatives and its equivalence with automata. Partial derivatives were introduced by Antimirov [24] as an alternative to Brzozowski derivatives, since it avoids quotient resulting REs with respect to ACUI axioms. Almeida et al. motivation is to use such formalization as a basis for a decision procedure for RE equivalence.

Ausaf et. al. [25] describe a formalization, in Isabelle/HOL [26], of the POSIX matching algorithm proposed by Sulzmann et.al. [19]. They give a constructive characterization of what a POSIX matching is and prove that such matching is unique for a given RE and string. No experiments with the verified algorithm are reported.

Ribeiro and Du Bois [27] describe the formalization of regular expression (RE) parsing algorithm that produces a bit representation of its parse tree in Agda. The algorithm computes bit-codes using Brzozowski derivatives and they prove that produced codes are equivalent to parse trees ensuring soundness and completeness w.r.t an inductive RE semantics. Like our work, Ribeiro and Du Bois developed a tool for RE matching and execute some experiments using it but they didn't consider partial derivatives to produce parsing evidence (in their case, bit-codes and proofs that such codes are equivalent to RE parse trees).

Lopes et. al. [28] describe an Idris formalization of a RE parsing tool using Brzozowski's derivatives. Like our work, they proved both soundness and completeness lemmas for the derivative operation and used data-types for denoting prefixes and substrings proofs for a given input RE and string. Lopes et. al. [28] also mention that they use natural numbers in Peano notation to represent alphabet symbols as their respective ASCII codes, instead of using Idris type for characters. According to the authors, the reason for this design choice is due to the way that Idris deals with propositional equality for primitive types, like Char. Equalities of values of these types only reduce on concrete primitive values; this causes computation of proofs to stop under variables whose type is a primitive one [28].

7 Conclusion

In this work, we describe a complete formalization of a derivative-based parsing for REs in Agda. Our tool supports algorithms based on Brzozowski's derivatives and Antimirov's partial derivatives to find all prefixes and substrings that match a given input RE. The developed formalization has 1145 lines of code, organized in 20 modules. We have proven 39 theorems and lemmas to complete the development. Most of them are immediate pattern matching functions over inductive datatypes and were omitted from this text for brevity. The complete Agda formalization, instructions on how to build and use it are available on the project's on-line repository [12].

As future work, we intend to work on the development of a certified program of greedy and POSIX RE parsing using derivatives [19, 20] and investigate ways to obtain a formalized but simple and efficient RE parsing tool.

Acknowledgements This work is supported by the CNPq – Brazil under grant No.: CNPq 426232/2016. This workd was funded by FAPEMIG — Fundação de Amparo a Pesquisa de Minas Gerais, Brazil.

References

[1] J. E. Hopcroft, R. Motwani, Rotwani, and J. D. Ullman, *Introduction to Automata Theory, Languages and Computability*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] M. E. Lesk and E. Schmidt, "Unix vol. ii," A. G. Hume and M. D. McIlroy, Eds. Philadelphia, PA, USA: W. B. Saunders Company, 1990, ch. Lex&Mdash;a Lexical Analyzer Generator, pp. 375–387. [Online]. Available: http://dl.acm.org/citation.cfm?id=107172.107193

[3] "GNU Grep home page," https://www.gnu.org/software/grep/.

[4] D. Firsov and T. Uustalu, "Certified parsing of regular languages," in *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, ser. Lecture Notes in Computer Science, G. Gonthier and M. Norrish, Eds., vol. 8307. Springer, 2013, pp. 98–113. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-03545-1_7

[5] ——, "Certified CYK parsing of context-free languages," *Journal of Logical and Algebraic Methods in Programming*, vol. 83, no. 5–6, pp. 459 – 468, 2014, the 24th Nordic Workshop on Programming Theory (NWPT 2012). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352220814000601

[6] N. A. Danielsson, "Total parser combinators," *SIGPLAN Not.*, vol. 45, no. 9, pp. 285–296, Sep. 2010. [Online]. Available: http://doi.acm.org/10.1145/1932681.1863585

[7] J. A. Brzozowski, "Derivatives of regular expressions," *J. ACM*, vol. 11, no. 4, pp. 481–494, Oct. 1964. [Online]. Available: http://doi.acm.org/10.1145/321239.321249

[8] S. Owens, J. Reppy, and A. Turon, "Regular-expression derivatives re-examined," *J. Funct. Program.*, vol. 19, no. 2, pp. 173–190, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1017/S0956796808007090

[9] M. Might, D. Darais, and D. Spiewak, "Parsing with derivatives: A functional pearl," *SIGPLAN Not.*, vol. 46, no. 9, pp. 189–195, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2034574.2034801

[10] S. Fischer, F. Huch, and T. Wilke, "A play on regular expressions: Functional pearl," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '10. New York, NY, USA: ACM, 2010, pp. 357–368. [Online]. Available: http://doi.acm.org/10.1145/1863543.1863594

[11] U. Norell, "Dependently typed programming in agda," in *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, ser. TLDI '09. New York, NY, USA: ACM, 2009, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/1481861.1481862

[12] "The verified grep tool - online repository." https://github.com/lives-group/vergrep.

[13] P. Martin-Löf, "An intuitionistic theory of types," in *Twenty-five years of constructive type theory (Venice, 1995)*, ser. Oxford Logic Guides. Oxford Univ. Press, New York, 1998, vol. 36, pp. 127–172.

[14] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)*. New York, NY, USA: Elsevier Science Inc., 2006.

[15] C. McBride and J. McKinna, "The view from the left," *J. Funct. Program.*, vol. 14, no. 1, pp. 69–111, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1017/S0956796803004829

[16] A. Stump, *Verified Functional Programming in Agda*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2016.

[17] V. Antimirov, "Partial derivatives of regular expressions and finite automaton constructions," *Theoretical Computer Science*, vol. 155, no. 2, pp. 291 – 319, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397595001824

[18] "Google Regular Expression Library - re2," https://github.com/google/re2.

[19] M. Sulzmann and K. Z. M. Lu, "POSIX regular expression parsing with derivatives," in *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, ser. Lecture Notes in Computer Science, M. Codish and E. Sumii, Eds., vol. 8475. Springer, 2014, pp. 203–220. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07151-0_13

[20] A. Frisch and L. Cardelli, "Greedy regular expression matching," in *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, ser. Lecture Notes in Computer Science, J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, Eds., vol. 3142. Springer, 2004, pp. 618–629. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27836-8_53

[21] M. Felleisen, M. B. Conrad, D. V. Horn, and E. S. of Northeastern University, *Realm of Racket: Learn to Program, One Game at a Time!* San Francisco, CA, USA: No Starch Press, 2013.

[22] H. D. Macedo and J. N. Oliveira, "Typing linear algebra: A biproduct-oriented approach," *CoRR*, vol. abs/1312.4818, 2013. [Online]. Available: http://arxiv.org/abs/1312.4818

[23] J. B. Almeida, N. Moreira, D. Pereira, and S. M. de Sousa, "Partial derivative automata formalized in coq," in *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Domaratzki and K. Salomaa, Eds., vol. 6482. Springer, 2010, pp. 59–68. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18098-9_7

[24] V. Antimirov, "Partial derivatives of regular expressions and finite automaton constructions," *Theoretical Computer Science*, vol. 155, no. 2, pp. 291 – 319, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397595001824

[25] F. Ausaf, R. Dyckhoff, and C. Urban, "POSIX lexing with derivatives of regular expressions (proof pearl)," in *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, ser. Lecture Notes in Computer Science, J. C. Blanchette and S. Merz, Eds., vol. 9807. Springer, 2016, pp. 69–86. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-43144-4_5

[26] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[27] R. Ribeiro and A. D. Bois, "Certified Bit-Coded Regular Expression Parsing," *Proceedings of the 21st Brazilian Symposium on Programming Languages - SBLP 2017*, pp. 1–8, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3125374.3125381

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