# Certified Derivative-Based Parsing of Regular Expressions

Rodrigo Ribeiro[a], Raul Lopes

*Dep. de Computação, Universidade Federal de Ouro Preto, ICEB, Campus Universitário Morro do Cruzeiro, Ouro Preto, Minas Gerais, Brasil*

Carlos Camarão

*Dep. de Ciência da Computação, Universidade Federal de Minas Gerais, Av. Antônio Carlos 6627, Belo Horizonte, Minas Gerais, Brasil*

*[a]Dep. de Computação e Sistemas, Universidade Federal de Ouro Preto, ICEA, João Monlevade, Minas Gerais, Brasil*

**Abstract**

We describe the formalization of a certified algorithms for regular expression parsing based on Brzozowski and Antimirov derivatives, in the dependently typed language Agda. The formalized algorithms produces proofs that an input string matches a given regular expression or a proof 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, and practical experiments were conducted with it.

*Keywords:* Certified algorithms, regular expressions; dependent types

## 1. Introduction

Parsing is the process of analysing if a string of symbols conforms to given rules, involving also, in computer science, formally specifying the rules in a grammar and also, either the construction of data that makes evident the rules that have been used to conclude that the string of symbols can be obtained from

*Corresponding author

*Email addresses:* raulfpl@gmail.com (Raul Lopes), camarao@dcc.ufmg.br (Carlos Camarão)

*URL:* rodrigo@decsi.ufop.br (Rodrigo Ribeiro)

the grammar rules, or else indication of an error, representative of 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 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 correct parsing algorithms [4, 5, 6]. This interest is motivated by the recent development of dependently typed languages. Such languages 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 his work on functional encoding of RE derivatives have renewed interest on its use for parsing [9, 10]. In this work, we provide a complete formalization of an algorithm for RE parsing using derivatives, as presented by [8], and describe a RE based search tool that has been developed by us, using the dependently typed language Agda [11].

More specifically, our contributions are:

- A formalization of Brzozowski derivatives based RE parsing in Agda. The certified algorithm presented 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 quotient derivatives with respect to ACUI axioms[1] in an implementation by Owens et al. [8], called "smart-constructors", and its proof of correctness. We give formal proofs

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

that smart constructors indeed preserve the language recognized by REs.

- A formalization of Antimirov's derivatives and its use to construct a RE parsing algorithm. The main difference between partial derivatives and Brzozowski's is that the former computes a set of RE's using set operators instead of "smart-constructors". Producing a set of RE's avoids the need of quotienting the resulting REs w.r.t. ACUI axioms.

This paper extends our SBLP 2016 paper [12] by formalizing a RE parsing algorithm using Antimirov's partial derivatives [13]. Also our original paper uses Idris [14] instead of Agda. This change was motivated by a modification in Idris totality checker that refuses some (correct and total) proofs that are both accepted by Agda's and Coq totality checkers. All source code produced in Idris, Agda and Coq, including the LaTeX source of this article, instructions on how to build and use it are avaliable on-line [15].

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, some of their properties and describe how to build a correct parsing algorithm from them. Section 5 comments on the usage of the certified algorithm to build a tool for RE-based search and present some experiments with it. Related work is discussed on Section 6. Section 7 concludes.

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 and point out where all details can be found in the source code.

## 2. An Overview of Agda

Agda is dependently-typed functional programming language based on Martin-Löef intuitionistic type theory [16]. This means, in particular, that it has very

few built-in types. In fact, only function types and the type of all types are built-in. Everything else is a user-defined type. The type Set, also known as $\mathsf{Set}_0$, is the type of all "small" types, such as Bool, String and List Bool. The type Set1 is the type of Set and "others like it", such as Set → Bool, String → Set, and Set → Set. There is in fact an infinite hierarchy of types of the form Set $l$, where $l$ is a universe level, roughly,a natural integer. This stratification of universes is need to keep Agda consistent as a logical theory [17].

An ordinary (non-dependent) function type is written $A \to B$ and a dependent one is written $(x : A) \to B$ 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)
```

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[2] we can interpret types as logical formulas and terms as proofs. An 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 no an evidence that such proof is impossible. Some functions used in our formalization use this type.

Dependently typed pattern matching is built by using the so-called **with** construct, that allows for matching intermediate values [18]. If the matched value has a dependent type, then its result can affect the form of other values. For example, consider the following code that defines a type for natural number parity. If the natural number is even, it can be represented as the sum of two equal natural numbers; if it is odd, it is equal to one plus the sum of two equal values. Pattern matching on a value of Parity $n$ allows to discover if $n = j + j$ or $n = S(k + k)$, for some $j$ and $k$ in each branch of **with**. Note that the value of $n$ is specialized accordingly, using information "learned" by the type-checker.

[2] Also known as Curry-Howard "isomorphism" [17].

```
data Parity : ℕ → Set where
  Even : Parity (n + n)
  Odd : Parity (S (n + n))

parity : (n : ℕ) → Parity n
parity =  -- definition omitted

natToBin : ℕ → List Bool
natToBin zero = []
natToBin k with (parity k)
  natToBin (j + j) | Even = false :: natToBin j
  natToBin (suc (j + j)) | Odd = true :: natToBin j
```

A detailed discussion about the Agda language is out of the scope of this paper. A tutorial on Agda is avaliable [11].

## 3. Regular Expressions

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

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

where $a$ is a symbol from the underlying alphabet. In our original Idris formalization, we describe symbols of an alphabet as a natural number in Peano notation (type $\mathbb{N}$), i.e. the symbol's numeric code. 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. Thus, we decide to use the inductive type $\mathbb{N}$ to represent the codes of alphabet symbols. In our Agda formalization, we represent alphabet symbols using Char type.

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

Using the datatype for RE syntax, we can define a relation for RL membership. Such relation can be understood as a parse tree (or a proof term) that a string, represented by a list of Char values, belongs to the language of a given RE.

The following datatype defines RE semantics inductively, i.e., each of its constructor specifies how to build a parse tree for some string and RE.

```
data _ ∈ ⟦_⟧ : List Char → Regex → Set where
  ϵ : [] ∈ ⟦ ϵ ⟧
  $_ : (c : Char) → $ c ∈ ⟦ $ c ⟧
  _ • _ ⇒ _ : 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 $\epsilon$ states that the empty string (denoted by the empty list []) is in the language of RE $\epsilon$. Parse tree for single characters are built with $ $a$, which says that the singleton string [ $a$ ] is in RL for $ $a$. Given parse trees for REs $l$ and $r$; $xs \in \llbracket\ l\ \rrbracket$ and $ys \in \llbracket\ r\ \rrbracket$, we can use constructor _ • _ ⇒ _ to build a parse tree for the concatenation of these REs. Constructor _ + L_ (_ + R_) creates a parse tree for $l + r$ from a parse tree from $l(r)$. 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 to formalize derivative based parsing. They consist of pattern-matching on proofs of

$\_ \in [\![\_]\!]$ 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 as a RE as another RE is due to Brzozowski [7] and it relies on a function (called $\nu$) that determines if some RE accepts or not the empty string:

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

Decidability of $\nu(e)$ is proved by function $\nu[\ e\ ]$, 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′ ]
ν[ e • e′ ] | yes pr | (yes pr1) = yes (pr • pr1 ⇒ refl)
ν[ e • e′ ] | yes pr | (no ¬pr1) = no (¬pr1 ∘ π₂ ∘ [] ∈ • − invert)
ν[ e • e′ ] | no ¬pr | pr1 = no (¬pr ∘ π₁ ∘ [] ∈ • − invert)
ν[ e + e′ ] with ν[ e ] | ν[ e′ ]
```

```
ν[ e + e′ ] | yes pr | pr1 = yes (e′ +L pr)
ν[ e + e′ ] | no ¬pr | (yes pr1) = yes (e +R pr1)
ν[ e + e′ ] | no ¬pr | (no ¬pr1) = no ([ ¬pr , ¬pr1 ] ∘ ∈ + − invert)
ν[ e ⋆ ] = yes ((e • e ⋆ +L ϵ) ⋆)
```

The $\nu[\_]$ definition uses several inversion lemmas about RE semantics. Lemma $[] \in \bullet - \mathsf{invert}$ states that if the empty string is in the language of $l \bullet r$ (where $l$ and $r$ are arbitrary RE's) then the empty string belongs to $l$ and $r$'s languages. Lemma $\in + - \mathsf{invert}$ 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, $\epsilon$ 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:

- For union:

$$1)\ e + \emptyset \approx e \qquad 2)\ \emptyset + e \approx e$$

- For concatenation:

$$1)\ e\,\emptyset \approx \emptyset \qquad 2)\ e\,\epsilon \approx e$$
$$3)\ \emptyset\,e \approx \emptyset \qquad 4)\ \epsilon\,e \approx e$$

- For Kleene star:

$$1)\ \emptyset^\star \approx \epsilon \qquad 2)\ \epsilon^\star \approx \epsilon$$

These axioms are kept as invariants using functions that preserve them while building REs. For union, we just need to worry when one parameter denotes the empty language RE ($\emptyset$):

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

In concatenation, we need to deal with the possibility of parameters being the empty RE or the empty string RE. If one is the empty language ($\emptyset$) the result is also the empty language. Since empty string RE is identity for concatenation, we return, as a result, the other parameter.

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

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

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

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

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

*Proof.* By case analysis on the structure of $e$ and $e'$. The only interesting cases are when one of the expressions is $\emptyset$. If $e = \emptyset$, then $\emptyset\ ‘+\ e' = e'$ and the desired result follows. The same reasoning applies for $e' = \emptyset$. □

**Lemma 2** (Completeness of union)**.** *For all REs $e$, $e'$ and all strings $xs$, if $xs \in [\![\ e + e'\ ]\!]$ holds then $xs \in [\![\ e\ ‘+\ e'\ ]\!]$ also holds.*

*Proof.* By case analysis on the structure of $e$, $e'$. The only interesting cases are when one of the REs is $\emptyset$. If $e = \emptyset$, we need to analyse the structure of

$xs \in [\![\ e + e'\ ]\!]$. The result follows directly or by contradiction using $xs \in [\![\ \emptyset\ ]\!]$. The same reasoning applies when $e' = \emptyset$. □

**Lemma 3** (Soundness of concatenation)**.** *For all REs $e$, $e'$ and all strings $xs$, if $xs \in [\![\ e\ `\bullet\ e'\ ]\!]$ holds then $xs \in [\![\ e \bullet e'\ ]\!]$ also holds.*

*Proof.* By case analysis on the structure of $e$, $e'$. The interesting cases are when $e$ or $e'$ are equal to $\epsilon$ or $\emptyset$. When some of the REs are equal to $\emptyset$, the result follows by contradiction. If one of the REs are equal to $\epsilon$ the desired result is immediate, from the proof term $xs \in [\![\ e\ `\bullet\ e'\ ]\!]$, using list concatenation properties. □

**Lemma 4** (Completeness of concatenation)**.** *For all REs $e$, $e'$ and all strings $xs$, if $xs \in [\![\ e \bullet e'\ ]\!]$ holds then $xs \in [\![\ e\ `\bullet\ e'\ ]\!]$ also holds.*

*Proof.* By case analysis on the structure of $e$, $e'$. The interesting cases are when $e$ or $e'$ are equal to $\epsilon$ or $\emptyset$. When some of the REs are equal to $\emptyset$, the result follows by contradiction. If one of the REs are equal to $\epsilon$ the desired result is immediate. □

**Lemma 5** (Soundness of Kleene star)**.** *For all REs $e$ and string $xs$, if $xs \in [\![\ e\ `\star\ ]\!]$ then $xs \in [\![\ e \star\ ]\!]$.*

*Proof.* Straightforward case analysis on $e$'s structure. □

**Lemma 6** (Completeness of Klenne star)**.** *For all REs $e$ and all strings $xs$, if $xs \in [\![\ e \star\ ]\!]$ holds then $xs \in [\![\ e\ `\star\ ]\!]$ also holds.*

*Proof.* Straightforward case analysis on $e$'s structure. □

All definitions of smart constructors and their properties are contained in `Smart.agda`, in the project's on-line repository [15].

*4.2. Brzozowski Derivatives and its Properties*

The derivative of a RE 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, which produces 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 ]
∂[ e • e′ , c ] | yes pr = (∂[ e , c ] ‘• e′) ‘+ ∂[ 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 derivative operation. Soundness of $\partial[\_,\_]$ ensures that if a string $xs$ is in $\partial[\ e\ ,\ x\ ]$'s

language, then $(x :: xs) \in [\![ e ]\!]$ holds. Completeness ensures that the other direction of implication holds.

**Theorem 1** (Soundness of derivative operation)**.** *For all RE $e$, string $xs$ and symbol $x$, if $xs \in [\![ \partial[ e , x ] ]\!]$ then $(x :: xs) \in [\![ e ]\!]$ holds.*

*Proof.* By induction on the structure of $e$, using the soundness lemmas for smart constructors and decidability of the emptiness test. □

**Theorem 2** (Completeness of derivative operation)**.** *For all RE $e$, string $xs$ and symbol $x$, if $(x :: xs) \in [\![ e ]\!]$ then $xs \in [\![ \partial[ e , x ] ]\!]$.*

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

Definitions and properties of Brzozowski derivatives are given in `Brzozowski.agda`, in the project's on-line repository [15].

### *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 as a method to construct a NFA (non-deterministic finite automata). The main insight of partial derivatives for building NFA's is building a set of RE's which, collectively accepts the same strings as Brzozowski derivatives. Algebraic properties of set operations ensures that ACUI equations holds. Below, we present function $\nabla_a(e)$ which computes the set of partial derivatives from a given RE $e$ and a symbol $a$.

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

Function $\nabla_a(e)$ uses the operator $S \odot e'$ which concatenates RE $e'$ at right of every $e \in S$:

$$S \odot e' = \{e \bullet e' \mid e \in S\}$$

Our agda implementation models sets as lists of regular expressions.

```
Regexes = List Regex
```

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

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

Definition of a function to compute partial derivatives for a given RE is defined as a direct translation of mathematical notation to Agda code:

```
∇[_,_] : Regex → Char → Regexes
∇[ ∅ , c ] = []
∇[ ϵ , c ] = []
∇[ $ x , c ] with x ≟ c
∇[ $ x , .x ] | yes refl = [ ϵ ]
∇[ $ x , c ] | no p = []
```

$$\nabla[\ e \bullet e'\ ,\ c\ ]\ \textbf{with}\ \nu[\ e\ ]$$
$$\nabla[\ e \bullet e'\ ,\ c\ ]\ |\ \text{yes}\ p\ =\ (e' \odot \nabla[\ e\ ,\ c\ ])\ +\!+\ \nabla[\ e'\ ,\ c\ ]$$
$$\nabla[\ e \bullet e'\ ,\ c\ ]\ |\ \text{no}\ \neg p\ =\ e' \odot \nabla[\ e\ ,\ c\ ]$$
$$\nabla[\ e + e'\ ,\ c\ ]\ =\ \nabla[\ e\ ,\ c\ ]\ +\!+\ \nabla[\ e'\ ,\ c\ ]$$
$$\nabla[\ e\star\ ,\ c\ ]\ =\ (e\star) \odot \nabla[\ e\ ,\ c\ ]$$

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

$$\begin{aligned}&\textbf{data}\ \_ \in \{\_\}\ :\ \text{List Char}\ \to\ \text{Regexes}\ \to\ \text{Set}\ \textbf{where}\\ &\quad \text{here}\ :\ s \in [\![\ e\ ]\!]\ \to\ s \in \{\ e :: es\ \}\\ &\quad \text{there}\ :\ s \in \{\ es\ \}\ \to\ s \in \{\ e :: es\ \}\end{aligned}$$

Essentially, a value of type $s \in \{\ S\ \}$ represents that $s$ is accepted by some RE in $S$. The next lemmas present the relation of $s \in \{\ S\ \}$ and list concatenation.

**Lemma 7** (Weakening left)**.** *For all sets of RE's $S$, $S'$ and string $s$, if $s \in \{\ S\ \}$ then $s \in \{\ S\ +\!+\ S'\ \}$.*

*Proof.* Straightforward induction on the derivation of $s \in \{\ S\ \}$. □

**Lemma 8** (Weakening right)**.** *For all sets of RE's $S$, $S'$ and string $s$, if $s \in \{\ S'\ \}$ then $s \in \{\ S\ +\!+\ S'\ \}$.*

*Proof.* Straightforward induction on the derivation of $s \in \{\ S'\ \}$. □

**Lemma 9.** *For all sets of RE's $S$, $S'$ and string $s$, if $s \in \{\ S\ +\!+\ S'\ \}$ then $s \in \{\ S\ \}$ or $s \in \{\ S'\ \}$.*

*Proof.* Induction on the derivation of $s \in \{\ S\ +\!+\ S'\ \}$ and case analysis on the structure of $S$ and $S'$. □

**Lemma 10.** *For all set of RE's $S$, RE $e$ and strings $s$, $s'$; if $s \in \{\ S\ \}$ and $s' \in [\![\ e\ ]\!]$ then $s\ +\!+\ s' \in \{\ e \odot S\ \}$.*

*Proof.* Induction on the derivation of $s \in \{\ S\ \}$. □

**Lemma 11.** *For all set of RE's $S$, RE $e$ and string $s$, if $s \in \{\ e \odot S\ \}$ then exists $s_1$ and $s_2$ s.t. $s \equiv s_1 ++ s_2$, $s_1 \in \{\ S\ \}$ and $s_2 \in [\![\ e\ ]\!]$.*

*Proof.* Induction on the derivation of $s \in \{\ e \odot S\ \}$. □

Using these previous results about $\_ \in \{\_\}$, we can enunciate the 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 $(a :: s) \in [\![\ e\ ]\!]$. Completeness theorems shows that the other direction of the soundness implication also holds.

**Theorem 3** (Soundness of partial derivative operation)**.** *For all symbol $a$, string $s$ and RE $e$, if $s \in \{\ \nabla[\ e\ ,\ a\ ]\ \}$ then $(a :: s) \in [\![\ e\ ]\!]$.*

*Proof.* Induction on $e$'s structure using Lemmas 9 and 11. □

**Theorem 4** (Completeness of partial derivative operation)**.** *For all symbol $a$, string $s$ and RE $e$, if $(a :: s) \in [\![\ e\ ]\!]$ then $s \in \{\ \nabla[\ e\ ,\ a\ ]\ \}$.*

*Proof.* Induction on $e$'s structure using Lemmas 7, 8 and 10. □

Definitions and properties of Antimirov's partial derivatives are given in `Antimirov.agda`, in the project's on-line repository [15].

### *4.4. Parsing*

When parsing a string using a given RE we are interested in discover what parts of input string are matched. Basically, RE parsing involves determine what prefixes and substring's of input match a given RE. For this, we define datatypes for representing when a RE $e$ matches a prefix or a substring of $xs$.

We say that RE $e$ matches a prefix of string $xs$ if there's exists strings $ys$ and $zs$ s.t. $xs \equiv ys ++ zs$ and $ys \in [\![\ e\ ]\!]$. Definition of IsPrefix datatype encode this concept. Datatype IsSubstring specifies when a RE $e$ matches a substring in $xs$, i.e. there's must exists strings $ys$, $zs$ and $ws$ s.t. $xs \equiv ys ++ zs ++ ws$ and $zs \in [\![\ e\ ]\!]$.

```
data IsPrefix (xs : List Char) (e : Regex) : Set where
  Prefix : ∀ (ys zs) → xs ≡ ys ++ zs → ys ∈ ⟦ e ⟧ → IsPrefix xs e
data IsSubstring (xs : List Char) (e : Regex) : Set where
  Substring : xs ≡ ys ++ zs ++ ws → zs ∈ ⟦ e ⟧ → IsSubstring xs e
```

Using these datatypes we can define some relevant properties of prefixes and substrings that are used in certified functions to compute them.

**Lemma 12** (Lemma ¬IsPrefix)**.** *For all RE* $e$*, if* $[\,] \in \llbracket\ e\ \rrbracket$ *is false then* IsPrefix $[\,]$ $e$ *is false.*

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

**Lemma 13** (Lemma ¬IsPrefix− ::)**.** *For all RE* $e$ *and string* $xs$*, if* $[\,] \in \llbracket\ e\ \rrbracket$ *is false and* IsPrefix $xs$ $\partial[\ e\ ,\ x\ ]$ *is false then* IsPrefix $(x :: xs)$ $e$ *is false.*

*Proof.* Immediate from IsPrefix definition and Theorem 2. □

**Lemma 14** (Lemma ¬IsSubstring)**.** *For all RE* $e$*, if* IsPrefix $[\,]$ $e$ *is false then* IsSubstring $[\,]$ $e$ *is false.*

*Proof.* Immediate from IsPrefix and IsSubstring definitions. □

**Lemma 15** (Lemma ¬IsSubstring− ::)**.** *For all strings* $xs$*, symbol* $x$ *and RE* $e$*, if* IsPrefix $(x :: xs)$ $e$ *is false and* IsSubstring $xs$ $e$ *is false then* IsSubstring $(x :: xs)$ $e$ *is false.*

*Proof.* Immediate from IsPrefix and IsSubstring definitions. □

Function IsPrefixDec decides if a given RE $e$ matches a prefix in $xs$ by induction on $xs$ structure using Lemmas 12, 13, 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 $x$ if RE $\partial[\ e\ ,\ x\ ]$ matches a prefix of the input string. If it 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 ¬IsPrefix– ::.

```
IsPrefixDec : ∀ (xs : List Char) (e : Regex) → 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 ]
IsPrefixDec (x :: xs) e | yes p = yes (Prefix [ ] (x :: xs) refl p)
IsPrefixDec (x :: xs) e | no ¬p with IsPrefixDec xs (∂[ e , x ])
IsPrefixDec (x :: xs) e | no ¬p | (yes (Prefix ys zs eq wit))
  = yes (Prefix (x :: ys) zs (cong (_ :: _ x) eq) (∂ – sound _ _ _ wit))
IsPrefixDec (x :: xs) e | no ¬pn | (no ¬p) = no (¬IsPrefix– :: ¬pn ¬p)
```

Function IsSubstringDec is also defined by induction on input strings structure using IsPrefixDec to check if it is possible to match a prefix of input using $e$. 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's a substring match, returning such substring or a proof that it does not exist.

```
IsSubstringDec : ∀ (xs : List Char) (e : Regex) → Dec (IsSubstring xs e)
IsSubstringDec [ ] e with ν[ e ]
IsSubstringDec [ ] e | yes p = yes (Substring [ ] [ ] [ ] refl p)
IsSubstringDec [ ] e | no ¬p = no (¬IsSubstring (¬IsPrefix ¬p))
IsSubstringDec (x :: xs) e with IsPrefixDec (x :: xs) e
IsSubstringDec (x :: xs) e | yes (Prefix ys zs eq wit)
  = yes (Substring [ ] ys zs eq wit)
IsSubstringDec (x :: xs) e | no ¬p with IsSubstringDec xs e
IsSubstringDec (x :: xs) e | no ¬p | (yes (Substring ys zs ws eq wit))
  = yes (Substring (x :: ys) zs ws (cong (_ :: _ x) eq) wit)
IsSubstringDec (x :: xs) e | 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 prefix and substring relation. Such lemmas are slighly modified versions of Lemmas 12 and 13 that consider the relation $\_ \in \{\_\}$ and are omitted for brevity.

Definitions and properties of functions for prefix and substring computation are given in folders `Prefix` and `Substring`, in the project's on-line repository [15].

## 5. Implementation Details and Experiments

From the algorithm formalized 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 and use Agda Standard Library and its support for calling Haskell functions throught foreign function interface.

In order to validade our tool (named verigrep — for verified Grep), we compare its performance with GNU Grep [3] (grep), Google regular expression library [19] (re2) and with Haskell RE parsing algorithms described in [10] (haskell-regexp). 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 use the same experiments as [20] using files formed by thousands of occurrences of symbol `a` were parsed, using the RE $(a + b + ab)^\star$; in the second, files with thousands of occurrences of `ab` were parsed using the same RE. Results are presented in Figures 1 and 2, respectively.

Our tool behaves poorly when compared with all other options considered. This inefficiency could be explained: 1) Our algorithm relies on the Brzozowski definition of 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

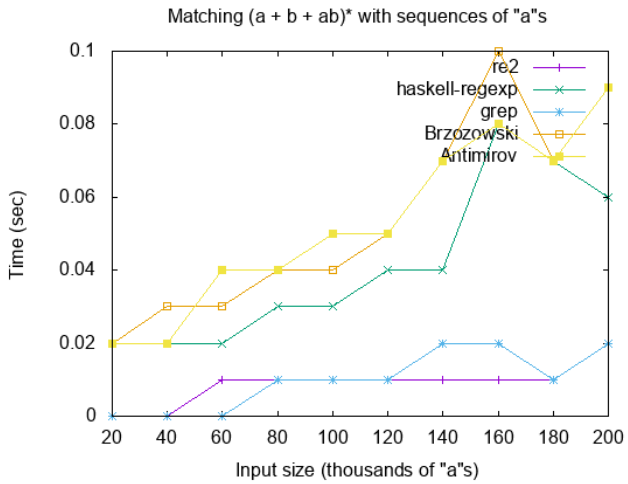


Figure 1: Results of experiment 1.

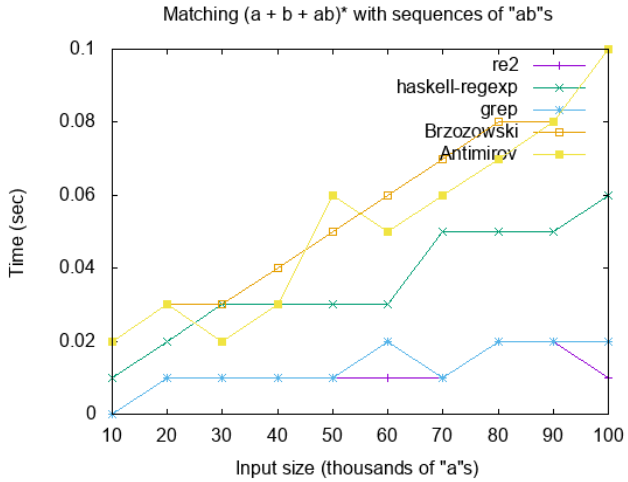


Figure 2: Results of experiment 2.

better data structures to represent sets and using appropriate disambiguation strategies like greedy parsing [21] and POSIX [20] would be able to improve the efficiency of our algorithm without sacrificing its correctness. We leave the formalization of disambiguation strategies and the use of more efficient data structures for future work.

## 6. Related Work

*Parsing with derivatives.* : 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] and no formal proof of correctness were presented.

Might et al. [9] report on the use of derivatives for parsing not only RLs but also context-free ones. He uses 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 by using the Racket programming language [22]. However, Might et al. do not present formal proofs related to the use of derivatives for CFGs.

Fischer et al. describes 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 [19]. This work also does not consider formal proofs of RE parsing.

An algorithm for POSIX RE parsing is described in [20]. 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 parsing algorithms.* : 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 non-deterministic finite automata (NFA) represented as a boolean matrix [4]. A matrix library based on some "block" operations [23] is developed and used Agda formalization of NFA-based parsing in Agda [11]. Compared to our work, a NFA-based formalization requires a lot more infrastructure (such as a Matrix library). No experiments with

the certified algorithm were reported.

Firsov describes an Agda formalization of a parsing algorithm that deals with any CFG (CYK algorithm) [24]. Bernardy et al. describe a formalization of another CFG parsing algorithm in Agda [25]: Valiant's algorithm [26], which reduces CFG parsing to boolean matrix multiplication. In both works, no experiment with formalized parsing algorithms were reported.

A certified LR(1) CFG validator is described in [27]. The formalized checking procedure verifies if CFG and a automaton match. They proved soundness and completeness of the validator in Coq proof assistant [28]. Termination of LR(1) automaton interpreter is ensured by imposing a natural number bound.

Formalization of a parser combinator library was the subject of Danielsson's work [6]. He built a library of parser combinators using coinduction and provide correctness proofs of such combinators.

Almeida et al. [29] describes a Coq formalization of partial derivatives and its equivalence with automata. Partial derivatives were introduced by Antimirov [30] 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.

Ridge [31] describes a formalization, in HOL4 theorem prover, of combinator parsing library. A parser generator for such combinators is described and a proof that generated parsers are sound and complete is presented. According to Ridge, preliminary results shows that parsers built using his generator are faster than those created by Happy parser generator [32].

Ausaf et.al. [33] describes a formalization, in Isabelle/HOL [34], of POSIX matching algorithm proposed by Sulzmann et.al. [20]. 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 veried algorithm are reported.

## 7. Conclusion

We have given a complete formalization of a derivative-based parsing for REs in Agda. To the best of our knowledge, this is the first work that presents a complete certification and that uses the certified program to build a tool for RE-based search.

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.

As future work, we intend to work on the development of a certified program of greedy and POSIX RE parsing using Brzozowski derivatives [20, 21] and investigate on ways to obtain a formalized but simple and efficient RE parsing tool.

*Acknowledgements:.* The first author thanks CNPq for financial support. Second author thanks Fundação de Amparo a Pesquisa de Minas Gerais (FAPEMIG) for financial support.

## References

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

[2] M. E. Lesk, E. Schmidt, Unix vol. ii, W. B. Saunders Company, Philadelphia, PA, USA, 1990, Ch. Lex&Mdash;a Lexical Analyzer Generator, pp. 375–387.
URL `http://dl.acm.org/citation.cfm?id=107172.107193`

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

[4] D. Firsov, T. Uustalu, Certified parsing of regular languages, in: G. Gonthier, M. Norrish (Eds.), Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December

11-13, 2013, Proceedings, Vol. 8307 of Lecture Notes in Computer Science, Springer, 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

[5] D. Firsov, T. Uustalu, Certified CYK parsing of context-free languages, J. Log. Algebr. Meth. Program. 83 (5-6) (2014) 459–468. doi:10.1016/j.jlamp.2014.09.002.
URL http://dx.doi.org/10.1016/j.jlamp.2014.09.002

[6] N. A. Danielsson, Total parser combinators, SIGPLAN Not. 45 (9) (2010) 285–296. doi:10.1145/1932681.1863585.
URL http://doi.acm.org/10.1145/1932681.1863585

[7] J. A. Brzozowski, Derivatives of regular expressions, J. ACM 11 (4) (1964) 481–494. doi:10.1145/321239.321249.
URL http://doi.acm.org/10.1145/321239.321249

[8] S. Owens, J. Reppy, A. Turon, Regular-expression derivatives re-examined, J. Funct. Program. 19 (2) (2009) 173–190. doi:10.1017/S0956796808007090.
URL http://dx.doi.org/10.1017/S0956796808007090

[9] M. Might, D. Darais, D. Spiewak, Parsing with derivatives: A functional pearl, SIGPLAN Not. 46 (9) (2011) 189–195. doi:10.1145/2034574.2034801.
URL http://doi.acm.org/10.1145/2034574.2034801

[10] S. Fischer, F. Huch, T. Wilke, A play on regular expressions: Functional pearl, in: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP '10, ACM, New York, NY, USA, 2010, pp. 357–368. doi:10.1145/1863543.1863594.
URL 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 Im-

plementation, TLDI '09, ACM, New York, NY, USA, 2009, pp. 1–2. doi:10.1145/1481861.1481862.
URL http://doi.acm.org/10.1145/1481861.1481862

[12] R. Lopes, R. Ribeiro, C. Camarão, Certified derivative-based parsing of regular expressions, in: Programming Languages — Lecture Notes in Computer Science 9889, Springer, 2015, pp. 95–109.

[13] V. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, Theoretical Computer Science 155 (2) (1996) 291 – 319. doi:http://dx.doi.org/10.1016/0304-3975(95)00182-4.
URL http://www.sciencedirect.com/science/article/pii/0304397595001824

[14] E. Brady, Idris, a general-purpose dependently typed programming language: Design and implementation, Journal of Functional Programming 23 (2013) 552–593. doi:10.1017/S095679681300018X.
URL http://journals.cambridge.org/article_S095679681300018X

[15] R. Lopes, R. Ribeiro, C. Camarão, Certified derivative based parsing of regular expressions — on-line repository, https://github.com/raulfpl/idrisregexp (2016).

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

[17] 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.

[18] C. McBride, J. McKinna, The view from the left, J. Funct. Program. 14 (1) (2004) 69–111. doi:10.1017/S0956796803004829.
URL http://dx.doi.org/10.1017/S0956796803004829

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

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

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

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

[23] H. D. Macedo, J. N. Oliveira, Typing linear algebra: A biproduct-oriented approach, CoRR abs/1312.4818.
URL http://arxiv.org/abs/1312.4818

[24] D. Firsov, T. Uustalu, Certified {CYK} parsing of context-free languages, Journal of Logical and Algebraic Methods in Programming 83 (5–6) (2014) 459 – 468, the 24th Nordic Workshop on Programming Theory (NWPT 2012). doi:http://dx.doi.org/10.1016/j.jlamp.2014.09.002.
URL http://www.sciencedirect.com/science/article/pii/S2352220814000601

[25] J. Bernardy, P. Jansson, Certified context-free parsing: A formalisation of

valiant's algorithm in agda, CoRR abs/1601.07724.
URL http://arxiv.org/abs/1601.07724

[26] L. G. Valiant, General context-free recognition in less than cubic time, J. Comput. Syst. Sci. 10 (2) (1975) 308–315. doi:10.1016/S0022-0000(75)80046-8.
URL http://dx.doi.org/10.1016/S0022-0000(75)80046-8

[27] J.-H. Jourdan, F. Pottier, X. Leroy, Validating lr(1) parsers, in: Proceedings of the 21st European Conference on Programming Languages and Systems, ESOP'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 397–416. doi:10.1007/978-3-642-28869-2_20.
URL http://dx.doi.org/10.1007/978-3-642-28869-2_20

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

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

[30] V. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, Theoretical Computer Science 155 (2) (1996) 291 – 319. doi:http://dx.doi.org/10.1016/0304-3975(95)00182-4.
URL http://www.sciencedirect.com/science/article/pii/0304397595001824

[31] T. Ridge, Simple, functional, sound and complete parsing for all context-free grammars, in: Proceedings of the First International Conference on

Certified Programs and Proofs, CPP'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 103–118. doi:10.1007/978-3-642-25379-9_10.
URL http://dx.doi.org/10.1007/978-3-642-25379-9_10

[32] Happy: The parser generator for Haskell., http://www.haskell.org/happy.

[33] 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 - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings, Vol. 9807 of Lecture Notes in Computer Science, Springer, 2016, pp. 69–86. doi:10.1007/978-3-319-43144-4_5.
URL http://dx.doi.org/10.1007/978-3-319-43144-4_5

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