Certified Derivative-Based Parsing of Regular Expressions

Raul Lopes[1], Rodrigo Ribeiro[2], and Carlos Camarão[3]

[1] DECOM, Universidade Federal de Ouro Preto (UFOP), Ouro Preto
`raulfpl@gmail.com`
[2] DECSI, Universidade Federal de Ouro Preto (UFOP), João Monlevade
`rodrigo@decsi.ufop.br`
[3] DCC, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte
`camarao@dcc.ufmg.br`

Abstract We describe the formalization of a certified algorithm for regular expression parsing based on Brzozowski derivatives, in the dependently typed language Idris. The formalized algorithm produces a proof 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 algorithm, and practical experiments were conducted with this tool.

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 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) [16], 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 [19] and string search utilities [15]. Since such tools are widely used and parsing is pervasive in computing, there is a growing interest on correct parsing algorithms [10,11,8]. 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 [27], “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 [25,13]. In this work, we provide a complete formalization of an algorithm for RE parsing using derivatives, as presented by [27], and describe a RE based search tool that has been developed by us, using the dependently typed language Idris.

More specifically, our contributions are:

– A formalization of derivative based regular expression parsing in Idris. The certified RE parsing 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[4] in an implementation by Owens et al. [27], called "smart-constructors", and its proof of correctness. We give formal proofs that smart constructors indeed preserve the language recognized by REs.

The rest of this paper is organized as follows. Section 2 presents a brief introduction to Idris. Section 3 describes the encoding of REs and its parse trees. In Section 4 we define 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 Idris Version 0.11, 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.

The complete Idris development, instructions on how to build and use it can be found at [21].

2 An Overview of Idris

Idris [5] is a dependently typed functional programming language that focus on supporting practical programs. Idris syntax is inspired by Haskell's with some minor differences. Unlike Haskell, Idris is strict by default, but lazy evaluation is supported through code annotations. Idris allows the definition of datatypes using traditional Haskell and a GADT-style syntax. The type of types is called `Type`, rather than $\star$[5]. Each instance of `Type` has an implicit level, inferred by the compiler. Levels are cumulative — everything in $\texttt{Type}_n$ is also in $\texttt{Type}_{n+1}$.

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

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

[5] In Haskell, types are classified using kinds [28] instead of universe levels. The kind of types is denoted by $\star$ and type operators have functional kinds: $\kappa \to \kappa'$, where κ and κ' are kinds. As an example, in Haskell, type `Bool` has kind $\star$ and the list type constructor has kind $\star \to \star$.

```
data Nat = Z | S Nat
data Vec : Nat -> Type -> Type where
    Nil : Vec Z a
    (::) : a -> Vec n a -> Vec (S n) a
```

Constructor `Nil` builds empty vectors. The cons-operator inserts a new element in front of a vector of n elements (of type `Vec n a`) and returns a value of type `Vec (S n) a`. 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 (S n) a`.

```
head : Vec (S n) a -> a
head (x :: xs) = x
```

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

In Idris, free variables that start with a lower-case letter are considered to be implicit arguments, i.e. arguments that can be automatically infered by the compiler. It is also possible to mark arguments as implicit by surrounding them in curly braces. In function `head`, both `n : Nat` and `a : Type` are implicit arguments; they could be explicitly annotated in `head`'s type as follows:

```
head : {a : Type} -> {n : Nat} -> Vec (S n) a -> a
```

Thanks to the propositions-as-types principle[6] we can interpret types as logical formulas and terms as proofs. An example is the representation of equality as the following Idris type:

```
data (=) : a -> b -> Type where
   Refl : x = x
```

This type is called propositional equality[7]. 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 : Type -> Type where
     Yes : p -> Dec p
     No : Not p -> Dec p
```

Constructor `Yes` stores a proof that property `P` holds and `No` an evidence that such proof is impossible (`Not` is an implication of falsity). Some functions used in our formalization use this type.

[6] Also known as Curry-Howard "isomorphism" [30].

[7] Readers who know type theory probably have noticed that this equality encoding corresponds to the so-called heterogeneous equality [23], which is used in the Idris Prelude. Detailed discussions about equality in type theory can be found in [32].

Dependently typed pattern matching is built by using the so-called `with` construct, that allows for matching intermediate values [24]. 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.

```
data Parity : Nat -> Type where
   Even : Parity (n + n)
   Odd  : Parity (S (n + n))

parity : (n : Nat) -> Parity n
parity = -- definition omitted

natToBin : Nat -> List Bool
natToBin Z = Nil
natToBin k with (parity k)
   natToBin (j + j)     | Even = False :: natToBin j
   natToBin (S (j + j)) | Odd  = True  :: natToBin j
```

A detailed discussion about the Idris language is out of the scope of this paper. A tutorial on Idris is available [17].

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 formalization, we describe symbols of an alphabet as a natural number in Peano notation (type `Nat`), 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 `Nat` to represent the codes of alphabet symbols, since computation of its equality proofs behaves as expected in other languages, like e.g. Agda [26].

Datatype `RegExp`, defined below, encodes RE syntax:

```
data RegExp : Type where
  Zero : RegExp
  Eps  : RegExp
  Chr  : Nat -> RegExp
  Cat  : RegExp -> RegExp -> RegExp
  Alt  : RegExp -> RegExp -> RegExp
  Star : RegExp -> RegExp
```

Constructors `Zero` and `Eps` denote respectively the empty language ($\emptyset$) and empty string (ϵ). Alphabet symbols are constructed using `Chr` constructor. Bigger REs are built using concatenation (`Cat`), union (`Alt`) 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 `Nat` values, belongs to the language of a given RE. Datatype `InRegExp` defines RE semantics inductively.

```
data InRegExp : List Nat -> RegExp -> Type where
  InEps : InRegExp [] Eps
  InChr : InRegExp [ a ] (Chr a)
  InCat : InRegExp xs l ->
          InRegExp ys r ->
          zs = xs ++ ys ->
          InRegExp zs (Cat l r)
  InAltL : InRegExp xs l ->
           InRegExp xs (Alt l r)
  InAltR : InRegExp xs r ->
           InRegExp xs (Alt l r)
  InStar : InRegExp xs (Alt Eps (Cat e (Star e))) ->
           InRegExp xs (Star e)
```

Each constructor of `InRegExp` datatype specifies how to build a parse tree for some string and RE. Constructor `InEps` states that the empty string (denoted by the empty list `[]`) is in the language of RE `Eps`. Parse tree for single characters are built with `InChr a`, which says that the singleton string `[ a ]` is in RL for `Chr a`. Given parse trees for REs `l` and `r`; `InRegExp xs l` and `InRegExp ys r`, we can use constructor `InCat` to build a parse tree for the concatenation of these REs. Constructor `InAltL` (`InAltR`) creates a parse tree for `Alt 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 `InRegExp` and are omitted for brevity.

4 Derivatives, Smart Constructors and Parsing

4.1 Preliminaries

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 ν) that determines if some RE accepts or not the empty string:

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

Decidability of $\nu(e)$ is proved by function `hasEmptyDec`, 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 Idris type of decidable propositions, `Dec P`.

```
hasEmptyDec : (e : RegExp) -> Dec (InRegExp [] e)
hasEmptyDec Zero = No (void . inZeroInv)
hasEmptyDec Eps = Yes InEps
hasEmptyDec (Chr c) = No inChrNil
hasEmptyDec (Cat e e') with (hasEmptyDec e)
  hasEmptyDec (Cat e e') | (Yes prf) with (hasEmptyDec e')
    hasEmptyDec (Cat e e') | (Yes prf) | (Yes prf')
          = Yes (InCat prf prf' Refl)
    hasEmptyDec (Cat e e') | (Yes prf) | (No contra)
          = No (contra . snd . inCatNil)
  hasEmptyDec (Cat e e') | (No contra)
          = No (contra . fst . inCatNil)
hasEmptyDec (Alt e e') with (hasEmptyDec e)
  hasEmptyDec (Alt e e') | (Yes prf)
          = Yes (InAltL prf)
  hasEmptyDec (Alt e e') | (No contra) with (hasEmptyDec e')
    hasEmptyDec (Alt e e') | (No contra) | (Yes prf)
          = Yes (InAltR prf)
    hasEmptyDec (Alt e e') | (No contra) | (No f)
          = No (void . either contra f . inAltNil)
hasEmptyDec (Star e)
          = Yes (InStar (InAltL InEps))
```

The `hasEmptyDec` definition uses several inversion lemmas about RE semantics. Lemma `inZeroInv` states that no word is in the language denoted by RE `Zero` and `inChrNil` states that the empty string (represented by an empty list) isn't in language denoted by RE `Chr c`, for some `c : Nat`. Inversion lemmas for concatenation and choice are similar.

4.2 Smart Constructors

Following Owens et. al. [27], 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 [16]. 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 (`Zero`):

```
(.|.) : RegExp -> RegExp -> RegExp
Zero .|. e = e
e .|. Zero = e
e .|. e'   = Alt 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 (`Zero`) the result is also the empty language. Since empty string RE is identity for concatenation, we return, as a result, the other parameter.

```
(.@.) : RegExp -> RegExp -> RegExp
Zero .@. e = Zero
Eps .@. e  = e
e .@. Zero = Zero
e .@. Eps  = e
e .@. e'   = Cat e e'
```

For Kleene star both `Zero` and `Eps` are replaced by `Eps`.

```
star : RegExp -> RegExp
star Zero = Eps
star Eps = Eps
star e = Star 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 `InRegExp` proofs.

Lemma 1 (Soundness of union)**.** *For all REs `e`, `e'` and all strings `xs`, if `InRegExp xs (e .|. e')` holds then `InRegExp xs (Alt 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 `Zero`. If `e = Zero`, then `Zero .|. e' = e'` and the desired result follows. The same reasoning applies for `e' = Zero`. □

Lemma 2 (Completeness of union)**.** *For all REs* `e`*,* `e'` *and all strings* `xs`*, if* `InRegExp xs (Alt e e')` *holds then* `InRegExp xs (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 `Zero`. If `e = Zero`, we need to analyse the structure of `InRegExp xs (Alt e e')`. The result follows directly or by contradiction using `InRegExp xs Zero`. The same reasoning applies when `e' = Zero`. □

Lemma 3 (Soundness of concatenation)**.** *For all REs* `e`*,* `e'` *and all strings* `xs`*, if* `InRegExp xs (e .@. e')` *holds then* `InRegExp xs (Cat e e')` *also holds.*

Proof. By case analysis on the structure of `e`, `e'`. The interesting cases are when `e` or `e'` are equal to `Eps` or `Zero`. When some of the REs are equal to `Zero`, the result follows by contradiction. If one of the REs are equal to `Eps` the desired result is immediate, from the proof term `InRegExp xs (e .@.e')`, using list concatenation properties. □

Lemma 4 (Completeness of concatenation)**.** *For all REs* `e`*,* `e'` *and all strings* `xs`*, if* `InRegExp xs (Cat e e')` *holds then* `InRegExp xs (e .@. e')` *also holds.*

Proof. By case analysis on the structure of `e`, `e'`. The interesting cases are when `e` or `e'` are equal to `Eps` or `Zero`. When some of the REs are equal to `Zero`, the result follows by contradiction. If one of the REs are equal to `Eps` the desired result is immediate, using the following fact:

```
InRegExp xs' e -> xs = xs' ++ [] -> InRegExp xs e
```

which asserts that if a strings `xs'` is in `e`'s language, then so is `xs' ++ []`. □

Lemma 5 (Soundness of Kleene star)**.** *For all REs* `e` *and string* `xs`*, if* `InRegExp xs (star e)` *then* `InRegExp xs (Star e)`*.*

Proof. Straightforward case analysis on `e`'s structure. □

Lemma 6 (Completeness of Klenne star)**.** *For all REs* `e` *and all strings* `xs`*, if* `InRegExpa xs (Star e)` *holds then* `InRegExp xs (star e)` *also holds.*

Proof. Straightforward case analysis on `e`'s structure. □

All definitions of smart constructors and their properties are contained in `SmartCons.idr`, in the project's on-line repository [21].

4.3 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:

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

This function has an immediate translation to Idris. Notice that the derivative function uses smart constructors to quotient result REs with respect to the equivalence axioms presented in Section 4.2 and RE emptiness test. In the symbol case (constructor `Chr`), function `decEq` is used, which produces an evidence for equality of two `Nat` values.

```
deriv : (e : RegExp) -> Nat -> RegExp
deriv Zero c = Zero
deriv Eps c = Zero
deriv (Chr c') c with (decEq c' c)
  deriv (Chr c) c  | Yes Refl = Eps
  deriv (Chr c') c | No nprf = Zero
deriv (Alt l r) c = (deriv l c) .|. (deriv r c)
deriv (Star e) c = (deriv e c) .@. (Star e)
deriv (Cat l r) c with (hasEmptyDec l)
  deriv (Cat l r) c | Yes prf = ((deriv l c) .@. r) .|. (deriv r c)
  deriv (Cat l r) c | No nprf = (deriv l c) .@. r
```

From this definition we prove the following important properties of derivative operation. Soundness of `deriv` ensures that if a string `xs` is in `deriv e x`'s language, then `InRegExp (x :: xs) 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* `InRegExp xs (deriv x e)` *then* `InRegExp (x :: xs) e`*.*

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* `InRegExp (x :: xs) e` *then* `InRegExp xs (deriv 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 derivatives are given in `Search.idr`, in the project's on-line repository [21].

4.4 Parsing

RE parsing with derivatives uses the following definition that extends $\partial_a(e)$ from a single symbol to a whole word by induction on the word structure:

$$\begin{aligned}\partial^\star_\epsilon(e) &= e\\ \partial^\star_{a\,w}(e) &= \partial^\star_w(\partial_a(e))\end{aligned}$$

We say that a string w is in e's language if $\partial^\star_w(e)$ is nullable, that is, if $\nu(\partial^\star_w(e)) = \epsilon$.

The Idris encoding of this function involves testing if the RE used for parsing is a prefix or a substring of the parsed string. Prefixes of a string are represented by datatype `Prefix e xs`, which expresses that a string parsed by RE `e` is a prefix of `xs`.

```
data Prefix : (e : RegExp) -> (xs : List Nat) -> Type where
  MkPrefix : (ys : List Nat)     ->
             (zs : List Nat)     ->
             (eq : xs = ys ++ zs) ->
             (re : InRegExp ys e) ->
             Prefix e xs
```

In order to state that some string `ys` is a prefix of `xs`, we need to build a proof that `ys` matches `e` and that it is indeed a prefix of `xs`, by providing an evidence that, for some `zs`, we have that `xs = ys ++ zs`.

A function for building prefixes just recurse over the structure of the input string, using derivatives. Definition of decidability of `Prefix e xs` is an immediate consequence of Theorem 1. Definitions and properties about prefixes can be found in file `Prefix.idr` in the source-code [21].

Substrings are represented by the type

```
data Substring : (e : RegExp) -> (xs : List Nat) -> Type where
  MkSubstring : (ys : List Nat)           ->
                (ts : List Nat)           ->
                (zs : List Nat)           ->
                (eq : xs = ys ++ ts ++ zs) ->
                (re : InRegExp ts e)       ->
                Substring e xs
```

that specifies that a string `ts` is a substring of `xs` if it is parsed by `e` and if there exist strings `ys` and `zs` such that `xs = ys ++ ts ++ zs`. Deciding if a RE parses a substring of some input is straightforward by recursion over the input string using prefix decidability. Definitions about substrings can be found in `Substring.idr` [21].

5 Implementation Details and Experiments

From the algorithm formalized we built a tool for RE parsing in the style of GNU Grep [15]. We have used Lightyear [20], Idris parser combinator library, for parsing RE syntax and to deal with file I/O; we have used Idris effects library [6], which relies on dependent types to provide safe side-effect usage.

In order to validade our tool (named iGrep — for Idris Grep), we compare its performance with GNU Grep [15] (grep), Google regular expression library [29] (re2) and with Haskell RE parsing algorithms described in [13] (haskell-regexp). We run RE parsing experiments on a machine with a Intel Core I7 1.7 GHz, 8GB RAM running Mac OS X 10.11.4; the results were collected and the median of several test runs was computed.

We use the same experiments as [31] 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.

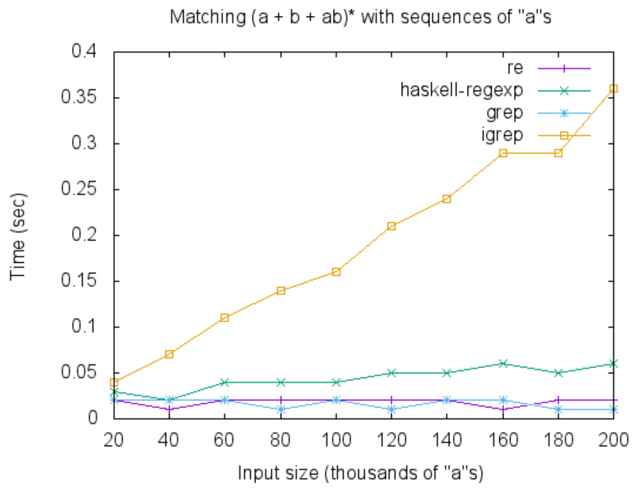

Figure 1. Results of experiment 1.

Our tool behaves poorly when compared with all other options considered. Possible causes for this inefficiency: 1) We represent alphabet symbols as natural numbers in Peano notation which has a costly equality test (linear on the term size); 2) our algorithm relies on the Brzozowski definition of RE parsing, which needs to quotient resulting REs. We believe that the use of disambiguation strategies like greedy parsing [14] and POSIX [31] would be able to improve the efficiency of our algorithm without sacrificing its correctness. The usage of these strategies can avoid the use of smart constructor to quotient equivalent REs. We leave the formalization of such disambiguation strategies for future work.

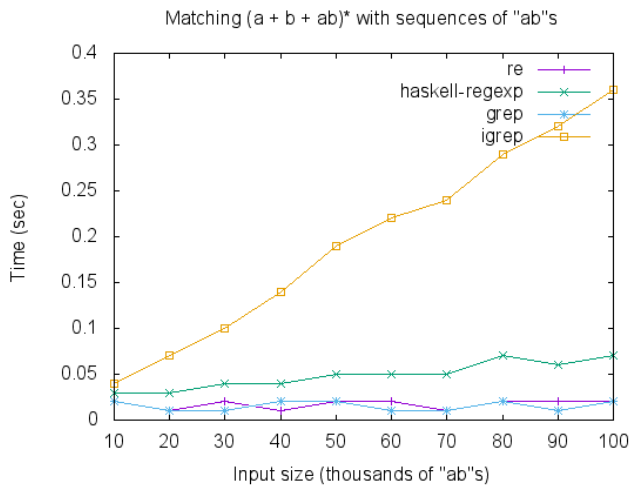

Figure 2. Results of experiment 2.

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 [27] and no formal proof of correctness were presented.

Might et al. [25] 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 [9]. 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 [13]. 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 [29]. This work also does not consider formal proofs of RE parsing.

An algorithm for POSIX RE parsing is described in [31]. 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 [10]. A matrix library based on some "block" operations [22] is developed and used Agda formalization of NFA-based parsing in Agda [26]. 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) [12]. Bernardy et al. describe a formalization of another CFG parsing algorithm in Agda [3]: Valiant's algorithm [33], 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 [18]. The formalized checking procedure verifies if CFG and a automaton match. They proved soundness and completeness of the validator in Coq proof assistant [4]. 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 [8]. He built a library of parser combinators using coinduction and provide correctness proofs of such combinators.

Almeida et al. [1] describes a Coq formalization of partial derivatives and its equivalence with automata. Partial derivatives were introduced by Antimirov [2] 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.

7 Conclusion

We have given a complete formalization of a derivative-based parsing for REs in Idris. 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 563 lines of code, organized in seven modules. We have proven 23 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 [31,14] and investigate on ways to obtain a formalized but simple and efficient RE parsing tool.

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

References

1. José Bacelar Almeida, Nelma Moreira, David Pereira, and Simão Melo de Sousa. Partial derivative automata formalized in coq. In Michael Domaratzki and Kai Salomaa, editors, *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers*, volume 6482 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2010.
2. Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.
3. Jean-Philippe Bernardy and Patrik Jansson. Certified context-free parsing: A formalisation of valiant's algorithm in agda. *CoRR*, abs/1601.07724, 2016.
4. Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.
5. Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013.
6. Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, New York, NY, USA, 2013. ACM.
7. Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.
8. Nils Anders Danielsson. Total parser combinators. *SIGPLAN Not.*, 45(9):285–296, September 2010.
9. Matthias Felleisen, M.D. Barski Conrad, David Van Horn, and Eight Students of Northeastern University. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press, San Francisco, CA, USA, 2013.
10. Denis Firsov and Tarmo Uustalu. Certified parsing of regular languages. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2013.
11. Denis Firsov and Tarmo Uustalu. Certified CYK parsing of context-free languages. *J. Log. Algebr. Meth. Program.*, 83(5-6):459–468, 2014.
12. Denis Firsov and Tarmo Uustalu. Certified {CYK} parsing of context-free languages. *Journal of Logical and Algebraic Methods in Programming*, 83(5–6):459 – 468, 2014. The 24th Nordic Workshop on Programming Theory (NWPT 2012).
13. Sebastian Fischer, Frank Huch, and Thomas Wilke. A play on regular expressions: Functional pearl. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 357–368, New York, NY, USA, 2010. ACM.
14. Alain Frisch and Luca Cardelli. Greedy regular expression matching. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*, volume 3142 of *Lecture Notes in Computer Science*, pages 618–629. Springer, 2004.
15. GNU Grep home page. `https://www.gnu.org/software/grep/`.

16. John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.
17. The Idris Tutorial. `http://docs.idris-lang.org/en/latest/tutorial/`.
18. Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating lr(1) parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems*, ESOP'12, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.
19. M. E. Lesk and E. Schmidt. Unix vol. ii. chapter Lex&Mdash;a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.
20. The Lightyear Idris Parsing Combinator Library. `https://github.com/ziman/lightyear/`.
21. Raul Lopes, Rodrigo Ribeiro, and Carlos Camarão. Certified derivative based parsing of regular expressions — on-line repository. https://github.com/raulfpl/idrisregexp, 2016.
22. Hugo Daniel Macedo and José Nuno Oliveira. Typing linear algebra: A biproduct-oriented approach. *CoRR*, abs/1312.4818, 2013.
23. Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, Department of Informatics, University of Edinburgh, 1999.
24. Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, January 2004.
25. Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. *SIGPLAN Not.*, 46(9):189–195, September 2011.
26. Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.
27. Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009.
28. Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.
29. Google Regular Expression Library - re2. `https://github.com/google/re2`.
30. Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA, 2006.
31. Martin Sulzmann and Kenny Zhuo Ming Lu. POSIX regular expression parsing with derivatives. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2014.
32. The Univalent Foundatiosn Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `http://homotopytypetheory.org/book/`, 2013.
33. Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, April 1975.