Certified Bit-Coded Regular Expression Parsing

Rodrigo Ribeiro
Universidade Federal de Ouro Preto
Ouro Preto, Minas Gerais — Brazil
rodrigo@decsi.ufop.br

André Du Bois
Universidade Federal de Pelotas
Pelotas, Rio Grande do Sul — Brazil
dubois@inf.ufpel.br

ABSTRACT

We describe the formalization of a regular expression (RE) parsing algorithm that produces a bit representation of its parse tree in the dependently typed language Agda. The algorithm computes bit-codes using Brzozowski derivatives and we prove that produced codes are equivalent to parse trees ensuring soundness and completeness w.r.t an inductive RE semantics. We include the certified algorithm in a tool developed by us, named verigrep, for regular expression based search in the style of the well known GNU grep. Practical experiments conducted with this tool are reported.

KEYWORDS

Certified algorithms, regular expressions, bit-codes, dependent types

ACM Reference format:
Rodrigo Ribeiro and André Du Bois. 2017. Certified Bit-Coded Regular Expression Parsing. In *Proceedings of SBLP 2017, Fortaleza, CE, Brazil, September 21–22, 2017,* 8 pages.
DOI: 10.1145/3125374.3125381

1 INTRODUCTION

Parsing is one of the most studied problems in computer science. It involves the process of checking if a string of symbols conforms to a given set of rules. Usually, parsing is preceded by the specification of rules in a formalism (e.g. a grammar) and, also, either the construction of data that makes evident which rules have been used to conclude that the string of symbols can be obtained from it or, otherwise, an indication of an error that represents the fact that the string of symbols cannot be generated.

In this work we are interested in the parsing problem for regular languages (RLs) [16], i.e. languages that can be recognized by nondeterministic 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 [18] and string search utilities [14]. Since such tools are widely used and parsing is pervasive in computing, there is a growing interest on certified parsing algorithms [7, 9, 10]. 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.

SBLP 2017, Fortaleza, CE, Brazil

DOI: 10.1145/3125374.3125381

In a previous work by one the authors [19], we described the formalization of an algorithm for parsing RE's based on derivatives. RE derivatives were introduced by Brzozowski [6] as an alternative method to compute a finite state machine that is equivalent to a given RE and to perform RE-based parsing. Owens et. al [27] reintroduce this concept, since "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 [12, 23].

Nielsen et. al. [24] introduce algorithms to build a compact representation of RE parse trees as a sequence of bits, without explicitly constructing the parse tree first and describe algorithms that simulate a finite state machine that output the required bit list. Sulzmann et. al. [33] designed a RE derivative-based algorithm that incrementally builds a list of bits instead of a tree as parsing result. In both works no machine checked proof was provided and Sulzmann et. al. informal proof "has some unfillable gaps", as pointed by Ausaf et. al. which proved that Sulzmann et. al. algorithm produces POSIX parse trees in Isabelle/HOL [3]. In this work we are interested in providing fully certified correctness proofs of Sulzmann et. al. derivative based parsing algorithm. The certified algorithm produces a bit sequence equivalent to traditional parse trees and we use it in a RE based search tool that has been developed by us, using the dependently typed language Agda [26]. Unlike Ausaf et. al., we do not provide mechanized proofs which ensure that our formalization produces POSIX parse trees. We leave such proof for further work.

More specifically, our contributions are:

- A formalization of bit-codes for RE parse trees, as presented in [24], and machine checkable proof of their properties.
- A formalization of bit-annotated regular expressions (BRE), its semantics and its soundness and completeness theorems w.r.t. standard RE semantics. We also relate produced bit sequences and RE semantics proving how they are related to each other.
- A formalization of Brzozowski derivatives for BRE and its soundness and completeness theorem w.r.t to BRE semantics.
- We build certified decision procedures for matching prefixes and substrings of BRE.

The rest of this paper is organized as follows. Section 2 presents a brief introduction to Agda. Section 3 describes the encoding of REs, its parse trees and their representation as bit-codes. In Section 3.2 we present BREs, its semantics and their relation with REs. In Section 4 we formalize a variant of Brzozowski derivatives for BREs, its properties and describe 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 presents some experiments with this tool. 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.

All source code produced, including the LaTeX source of this article, are avaliable on-line [29].

2 AN OVERVIEW OF AGDA

Agda is a dependently-typed functional programming language based on Martin-Löf intuitionistic type theory [21]. Function types and an infinite hierarchy of types of types, 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 List Bool. The type Set_1 is the type of Set and "others like it", such as Set → Bool, String → Set, and Set → Set. We have that Set l is an element of the type Set $(l + 1)$, for every $l \geqslant 0$. This stratification of types is used to keep Agda consistent as a logical theory [31].

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 values 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 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[1] we can interpret types as logical formulas and terms as proofs. An example is the representation of equality as the following Agda type:

[1] Also known as Curry-Howard "isomorphism" [31].

```
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 ¬ 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).

A useful data type in dependently typed languages is the so-called dependent-product, or Σ-types, that generalizes cartesian products. A possible definition of dependent products in Agda is as follows:

```
data Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b) where
  _,_ : (x : A) → B x → Σ A B
```

A value of type Σ A B corresponds to a pair formed by a value x of type A and a value of type B x. Note that the second component type depends on the value of the first component. Under proposition-as-types interpretation, dependent products are equivalent to existential quantification, since values of dependent products are formed by a value $x : A$, which can be understood as a witness of existential quantification, and a value B x, which represents the proof that x holds for B. Agda standard library represents existential quantification as a dependent product in which the first component is an implicit parameter:

```
∃ : ∀ {a b} {A : Set a} → (A → Set b) → Set (a ⊔ b)
∃ = Σ _
```

In some functions involving dependent products we use wilcards "_" to avoid the need to explicitly write their values or even completely omit, since Agda type checker can infer them. Finally, we represent projections on dependent products as π_1 and π_2 which recover the first and second component of product types, repectively.

Dependently typed pattern matching is built by using the so-called **with** construct, that allows for matching intermediate values [22]. 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 : ℕ → Set where
  Even : ∀ {n : ℕ} → Parity (n + n)
```

```
  Odd : ∀ {n : ℕ} → 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
```

For further information about Agda, see [26, 32].

3 REGULAR EXPRESSIONS

3.1 Standard Regular Expressions

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

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

where a is any symbol from the underlying alphabet. In our Agda formalization, we represent alphabet symbols using 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. Bigger REs are built using concatenation ($\bullet$), union ($+$) and Kleene star ($\star$).

The following datatype defines RE semantics inductively.

```
data _∈⟦_⟧ : List Char → Regex → Set where
  ϵ : [ ] ∈⟦ ϵ ⟧
  $_ : (c : Char) → [ c ] ∈⟦ $ c ⟧
  _•_ : xs ∈⟦ l ⟧ → ys ∈⟦ r ⟧ → (xs ++ ys) ∈⟦ 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 membership proofs for REs l and r, $xs \in \llbracket\ l\ \rrbracket$ and $ys \in \llbracket\ r\ \rrbracket$, constructor _•_ can be used to build a proof for the concatenation of these REs. Constructor _+L_ (_+R_) creates a membership proof for $l + r$ from a proof from l (r). Semantics for Kleene star is built using the following well known equivalence of REs: $e^\star = \epsilon + e\,e^\star$. Notice that we use the same constructor names both in RE syntax and in its semantics. Agda also allows overloading of data-constructors names.

Several inversion lemmas about RE parsing relation are necessary for derivative-based parsing formalization. They consist of pattern-matching on proofs of _∈⟦_⟧ and are omitted for brevity.

One way to look at RE parsing is to intepret parse trees as terms whose type is a RE [13, 24]. To ensure the consistency between a tree and its RE type, we use an indexed data-type, an usual approach in dependently typed languages [26].

```
data Tree : Regex → Set where
  ϵ : Tree ϵ
  $_ : (c : Char) → Tree ($ c)
  inl : ∀ (r : Regex) (tl : Tree l) → Tree (l + r)
  inr : ∀ (l : Regex) (tr : Tree r) → Tree (l + r)
  _•_ : ∀ (tl : Tree l) (tr : Tree r) → Tree (l • r)
  star[] : Tree (l ⋆)
  star–:: : Tree l → Tree (l ⋆) → Tree (l ⋆)
```

Each constructor of Tree specifies how to build a parse tree for a RE. As an example, value inl ϵ (($ 'a') • ($ 'b') denotes a parse tree for RE $ab + \epsilon$.

The relation between RE semantics and its parse trees are formalized by functions flat and unflat. The flat function builds a membership proof from a given parse tree by recursion on its structure. At each step, flat returns a pair formed by a string and its RE membership proof. Matched strings can be recovered from parse trees by concatenating values in their leaves (that would be the empty string or a single character).

```
flat : Tree e → ∃ (λ xs → xs ∈⟦ e ⟧)
flat ϵ = [ ] , ϵ
flat ($ c) = [ c ] , ($ c)
flat (inl r t) with flat t
...| xs , prf = _ , r +L prf
flat (inr l t) with flat t
...| xs , prf = _ , l +R prf
flat (t • t′) with flat t | flat t′
...| xs , prf | ys , prf′ = _ , (prf • prf′)
flat star[] = [ ] , (_ +L ϵ) ⋆
flat (star–:: t t′) with flat t | flat t′
...| xs , prf | ys , prf′ = _ , (_ +R (prf • prf′)) ⋆
```

Note that our definition of flat ensures, by construction, that the string produced by a parse tree is in its RE language. Such property is stated in Nielsen et. al. paper as a theorem (cf. Theorem 2.1 [24]).

Function unflat builds parse trees from RE membership proofs straightforwardly:

```
unflat : xs ∈⟦ e ⟧ → Tree e
unflat ϵ = ϵ
unflat ($ c) = $ c
unflat (prf • prf′) = unflat prf • unflat prf′
unflat (r +L prf) = inl r (unflat prf)
unflat (l +R prf) = inr l (unflat prf)
unflat ((_ +L ϵ) ⋆) = star[]
unflat ((_ +R (prf • prf′)) ⋆) = star–:: (unflat prf) (unflat prf′)
```

The next lemmas state that functions flat and unflat are inverses and they are proved by induction on t and $xs \in \llbracket\ e\ \rrbracket$, respectively.

Lemma 1. *Let e be a RE and t : Tree e a parse tree. Then* unflat (π_2 (flat t)) $\equiv t$.

Lemma 2. *Let xs be a string and e a RE s.t.* $xs \in [\![e]\!]$. *Then* flat (unflat *prf*) ≡ (*xs* , *prf*).

3.2 Bit-codes for RE Parse Trees

We follow the encoding of Nielsen et. al. [24] that uses bit marks to register which branch was chosen in a parse tree for a union operator, +, and the beginning and end of matches that correspond to a Kleene star. Note that no marking is needed for concatenation and single character since coding and decoding of bit sequences are directed by the underlining RE.

We represent bit sequences using lists whose elements are of type Bit.

```
data Bit : Set where
  0b 1b : Bit
```

Not every bit list corresponds to a valid parse tree. Data type IsCode represents an inductive predicate that holds when a bit list *bs* denotes a valid parse tree for some RE *e*.

```
data _IsCode_ : List Bit → Regex → Set where
  ϵ : [ ] IsCode ϵ
  $_ : (c : Char) → [ ] IsCode ($ c)
  inl : xs IsCode l → (0b :: xs) IsCode (l + r)
  inr : xs IsCode r → (1b :: xs) IsCode (l + r)
  _•_ : xs IsCode l → ys IsCode r → (xs ++ ys) IsCode (l • r)
  star[] : [ 1b ] IsCode (l ⋆)
  star–:: : xs IsCode l → xss IsCode (l ⋆) →
    (0b :: xs ++ xss) IsCode (l ⋆)
```

The empty string and single character RE are both represented by empty bit lists. Codes for RE $l \bullet r$ are built by concatenating codes of l and r. In RE union operator, +, the bit 0_b marks that the parse tree for $l + r$ is built from l's and bit 1_b that it is built from r's. For the Kleene star, we use bit 1_b to denote the parse tree for the empty string and bit 0_b to begin matchings of l in a parse tree for $l \star$.

Function code builds a bit representation for a parse tree together with a proof that the produced bit string is a valid tree representation.

```
code : Tree e → ∃ (λ bs → bs IsCode e)
code ϵ = [ ] , ϵ
code ($ c) = [ ] , ($ c)
code (inl r t) with code t
...| ys , pr = 0b :: ys , inl r pr
code (inr l t) with code t
...| ys , pr = 1b :: ys , inr l pr
code (t • t′) with code t | code t′
...| xs , pr | ys , pr′ = xs ++ ys , pr • pr′
code star[] = 1b :: [ ] , star[]
code (star–:: t ts) with code t | code ts
...| xs , pr | xss , prs = (0b :: xs ++ xss) , star–:: pr prs
```

Next we present function decode which generates a parse tree from its correspondent bit code.

```
decode : ∃ (λ bs → bs IsCode e) → Tree e
decode (_ , ϵ) = ϵ
decode (_ , ($ c)) = $ c
decode (_ , (inl r pr)) = inl r (decode (_ , pr))
decode (_ , (inr l pr)) = inr l (decode (_ , pr))
decode (_ , (pr • pr′)) with decode (_ , pr) | decode (_ , pr′)
...| bs1 , pr1 | bs2 , pr2 = pr1 • pr2
decode star[] = (_ +L ϵ) ⋆
decode (star–:: pr pr′) with decode (_ , pr) | decode (_ , pr′)
...| pr1 | pr2 = (_ +R (pr1 • pr2)) ⋆
```

As one might expect, decoding a bit sequence produced by the code function will produce the original parse tree.

Theorem 1. *Let t* : Tree *e for some RE e. Then, it holds that* decode (code *t*) ≡ *t*.

Building bit codes from parse trees does not present any advantage, since after building parse trees we will need to traverse them in order to generate codes. A better strategy is to incrementally build bit-codes while parsing a given RE. In next sections, we describe the formalization of this approach, due to Sulzmann et. al. [33].

3.3 Bit-annotated Regular Expressions

Intuitively, BRE just attach a list of bits *bs* to every non-empty regular expression. Type BitRegex defines the syntax of BRE, which just associates a list of bits to a BRE.

```
data BitRegex : Set where
  empty : BitRegex
  eps   : List Bit → BitRegex
  char  : List Bit → (c : Char) → BitRegex
  choice : List Bit → BitRegex → BitRegex → BitRegex
  cat   : List Bit → BitRegex → BitRegex → BitRegex
  star  : List Bit → BitRegex → BitRegex
```

Conversion between Regex and BitRegex types is done by functions internalize and erase. First, we define an auxiliary function, fuse, which attachs a bit code to the top-most position of a BRE.

```
fuse : List Bit → BitRegex → BitRegex
fuse bs empty = empty
fuse bs (eps x) = eps (bs ++ x)
fuse bs (char x c) = char (bs ++ x) c
fuse bs (choice x e e′) = choice (bs ++ x) e e′
fuse bs (cat x e e′) = cat (bs ++ x) e e′
fuse bs (star x e) = star (bs ++ x) e
```

Function internalize converts a standard RE into BRE by inserting empty bit lists on it.

```
internalize : Regex → BitRegex
internalize ∅ = empty
internalize ϵ = eps [ ]
internalize ($ x) = char [ ] x
internalize (e • e′) = cat [ ] (internalize e) (internalize e′)
internalize (e + e′)
  = choice [ ] (fuse [ 0b ] (internalize e))
    (fuse [ 1b ] (internalize e′))
internalize (e ⋆) = star [ ] (internalize e)
```

Function erase is straightforwardly defined by recursion.

```
erase : BitRegex → Regex
erase empty = ∅
erase (eps x) = ϵ
erase (char x c) = $ c
erase (choice x e e′) = erase e + (erase e′)
erase (cat x e e′) = erase e • (erase e′)
erase (star x e) = (erase e) ⋆
```

The relation between these functions is expressed by the following lemmas, which are proved by induction on the structure of e.

Lemma 3. *For all e and bs,* erase (fuse bs (internalize e)) ≡ e.

Lemma 4. *Let e : Regex. Then,* erase (internalize e) ≡ e

Next, we define an alternative inductive semantics for BRE and show that it is sound and complete w.r.t. standard RE semantics. A value of type $xs \in \langle\ e\ \rangle$ states that string xs is in the language denoted by BitRegex e.

```
data _ ∈ ⟨_⟩ : List Char → BitRegex → Set where
  eps : (bs : List Bit) → [ ] ∈ ⟨ eps bs ⟩
  char : (bs : List Bit) (c : Char) → [ c ] ∈ ⟨ char bs c ⟩
  inl : (r : BitRegex) bs → xs ∈ ⟨ l ⟩ → xs ∈ ⟨ choice bs l r ⟩
  inr : (l : BitRegex) bs → xs ∈ ⟨ r ⟩ → xs ∈ ⟨ choice bs l r ⟩
  cat : (bs : List Bit) → xs ∈ ⟨ l ⟩ → ys ∈ ⟨ r ⟩ →
        (xs ++ ys) ∈ ⟨ cat bs l r ⟩
  star[] : (bs : List Bit) → [ ] ∈ ⟨ star bs l ⟩
  star– :: : bs → (x :: xs) ∈ ⟨ l ⟩ → xss ∈ ⟨ star [ ] l ⟩ →
    (x :: xs ++ xss) ∈ ⟨ star bs l ⟩
```

The relation between BitRegex and Regex semantic is specified by the following theorem, proved by induction on the derivation of $xs \in [\![\ e\ ]\!]$.

Theorem 2. *Let e : Regex and xs : List Char. Then, $xs \in [\![\ e\ ]\!]$, if and only if, $xs \in \langle$ internalize $e\ \rangle$.*

4 DERIVATIVES 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 [6]. 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 $\nu[\ e\ ]$, which is defined by induction over the structure of the input BRE e and returns a proof that the empty string is accepted or not, using Agda type of decidable propositions, Dec P.

```
ν[_] : ∀ (e : BitRegex) → Dec ([ ] ∈ ⟨ e ⟩)
ν[ empty ] = no (λ ())
ν[ eps bs ] = yes eps
ν[ char bs x ] = no (λ ())
ν[ cat bs e e′ ] with ν[ e ] | ν[ e′ ]
ν[ cat bs e e′ ] | yes pr | (yes pr1)
  = yes (cat bs pr pr1)
ν[ cat bs e e′ ] | yes pr | (no ¬pr1)
  = no (¬pr1 ∘ π2 ∘ [] ∈ • – invert)
ν[ cat bs e e′ ] | no ¬pr | pr1
  = no (¬pr ∘ π1 ∘ [] ∈ • – invert)
ν[ choice bs e e′ ] with ν[ e ] | ν[ e′ ]
ν[ choice bs e e′ ] | yes pr | pr1 = yes (e′ +L pr)
ν[ choice bs e e′ ] | no ¬pr | (yes pr1) = yes (e +R pr1)
ν[ choice bs e e′ ] | no ¬pr | (no ¬pr1)
  = no ([ ¬pr , ¬pr1 ] ∘ ∈ + – invert)
ν[ star bs e ] = yes star[]
```

Function $\nu[\ e\ ]$ is an immediate translation of $\nu(e)$, as defined by Brzozowski, to Agda code and it uses several inversion lemmas about BRE semantics. Lemma [] ∈ • – invert states that if the empty string is in the language of cat $bs\ l\ r$ (where l and r are arbitrary BRE's) then the empty string belongs to l and r's languages. Lemma ∈ + – invert is defined similarly for choice.

4.1 Derivatives for Bit-annotated REs and its Properties

Following Sulzmann [33], we define a function mkEps that builds the bit code for a nullable BRE, (i.e. a BRE s.t. $[\ ] \in \langle\ e\ \rangle$).

```
mkEps : [ ] ∈ ⟨ t ⟩ → List Bit
mkEps (eps bs) = bs
mkEps (inl br bs pr) = bs ++ mkEps pr
mkEps (inr bl bs pr) = bs ++ mkEps pr
mkEps (cat bs pr pr′) = bs ++ mkEps pr ++ mkEps pr′
mkEps (star[] bs) = bs ++ [ 1b ]
mkEps (star– :: bs pr pr′ x) = bs ++ [ 1b ]
```

Next we define the derivative operation on BREs in Agda. The difference betwen this definition and standard Brzozowski derivatives [6] is that the former inserts parse tree information in terms of bit annotations. For example, consider cat $bs\ l\ r$ where $[\ ] \in \langle\ l\ \rangle$, additional parse information is built from a nullability test result using functions mkEps and fuse. For the Kleene star operation we record the start of a new iteration fusing [0_b] and we mark a start of a new matching iteration by attaching the empty list in star [ ] e.

```
∂[_,_] : BitRegex → Char → BitRegex
∂[ empty , c ] = empty
∂[ eps bs , c ] = eps bs
∂[ char bs c , c′ ] with c ≟ c′
...| yes refl = eps bs
...| no prf = empty
∂[ cat bs e e′ , c ] with ν[ e ]
∂[ cat bs e e′ , c ] | yes pr
  = choice bs (cat bs ∂[ e , c ] e′) (fuse (mkEps pr) ∂[ e′ , c ])
```

```
∂[ cat bs e e′ , c ] | no ¬pr = cat bs ∂[ e , c ] e′
∂[ choice bs e e′ , c ] = choice bs (∂[ e , c ]) (∂[ e′ , c ])
∂[ star bs e , c ]
  = cat bs (fuse [ 0b ] ∂[ e , c ]) (star [ ] 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 $(x :: xs) \in \langle\ e\ \rangle$ holds. Completeness ensures that the other direction of the implication holds. Both are proved by induction on the structure of e.

Theorem 3 (Derivative operation soundness). *For all BREs e, all strings xs and all symbols x, if $xs \in \langle\ \partial[\ e\ ,\ x\ ]\ \rangle$ holds then $(x :: xs) \in \langle\ e\ \rangle$ holds.*

Theorem 4 (Derivative operation completeness). *For all BREs e, all strings xs and all symbols x, if $(x :: xs) \in \langle\ e\ \rangle$ holds then $xs \in \langle\ \partial[\ e\ ,\ x\ ]\ \rangle$ holds.*

4.2 Parsing

RE parsing involves determining which prefixes and substrings of the input string match a given RE. For this, we define datatypes that represent the fact that a given BRE matches a prefix or a substring of a given string.

We say that BRE e matches a prefix of string xs if there exist strings ys and zs such that $xs \equiv ys$ ++ zs and $ys \in \langle\ e\ \rangle$. Definition of IsPrefix datatype encode this concept. Datatype IsSubstr specifies when a BRE e matches a substring in xs: there must exist strings ys, zs and ws such that $xs \equiv ys$ ++ zs ++ ws and $zs \in \langle\ e\ \rangle$ hold.

```
data IsPrefix (xs : List Char) (e : BitRegex) : Set where
  Prefix : xs ≡ ys ++ zs → ys ∈ ⟨ e ⟩ → IsPrefix xs e

data IsSubstr (xs : List Char) (e : BitRegex) : Set where
  Substr : xs ≡ ys ++ zs ++ ws → zs ∈ ⟨ e ⟩ → IsSubstr xs e
```

Using these datatypes we can define the following relevant properties of prefixes and substrings that are used to fill proof obligations present in decidability tests. All these lemmas are direct consequences of prefix and substring definitions.

Lemma 5 (Lemma ¬IsPrefix). *For all BREs e, if $[\ ] \in \langle\ e\ \rangle$ does not hold then neither does IsPrefix [ ] e.*

Lemma 6 (Lemma ¬IsPrefix– ::). *For all BREs e and all strings xs, if $[\ ] \in \langle\ e\ \rangle$ and IsPrefix xs $\partial[\ e\ ,\ x\ ]$ do not hold then neither does IsPrefix $(x :: xs)$ e.*

Lemma 7 (Lemma ¬IsSubstring). *For all BREs e, if IsPrefix [ ] e does not hold then neither does IsSubstr [ ] e.*

Lemma 8 (Lemma ¬IsSubstring– ::). *For all strings xs, all symbols x and all BREs e, if IsPrefix $(x :: xs)$ e and IsSubstr xs e do not hold then neither does IsSubstr $(x :: xs)$ e.*

Function IsPrefixDec decides if a given BRE e matches a prefix in xs by induction on the structure of xs, using Lemmas 5, 6, decidable emptyness test $\nu[\_]$ and Theorem 3. 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 BRE $\partial[\ e\ ,\ x\ ]$ 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 ¬IsPrefix– ::.

```
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 ]
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 IsSubstrDec is also defined by induction on the structure of the input string e, 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.

```
IsSubstrDec : ∀ xs e → Dec (IsSubstr xs e)
IsSubstrDec [ ] e with ν[ e ]
IsSubstrDec [ ] e | yes p = yes (Substr [ ] [ ] [ ] refl p)
IsSubstrDec [ ] e | no ¬p = no (¬IsSubstring (¬IsPrefix ¬p))
IsSubstrDec (x :: xs) e with IsPrefixDec (x :: xs) e
IsSubstrDec (x :: xs) e | yes (Prefix ys zs eq wit)
  = yes (Substr [ ] ys zs eq wit)
IsSubstrDec (x :: xs) e | no ¬p with IsSubstrDec xs e
IsSubstrDec (x :: xs) e | no ¬p | (yes (Substr ys zs ws eq wit))
  = yes (Substr (x :: ys) zs ws (cong (_ :: _ x) eq) wit)
IsSubstrDec (x :: xs) e | no ¬p1 | (no ¬p)
  = no (¬IsSubstring– :: ¬p1 ¬p)
```

5 IMPLEMENTATION DETAILS AND EXPERIMENTS

We include the formalized algorithm in a tool for RE parsing developed by us, named verigrep, in the style of GNU Grep [14]. 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 [14] (grep), Google regular expression library re2 [28] and Haskell RE parsing algorithms haskell-regexp, described in [12]. The experiments consider three distinct algorithms implemented in verigrep: RE parsing using Brzozowski derivatives, Antimirov partial derivatives and bit-coded parsing as described in this work. 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 those used in [33]; 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 Figures 1 and 2, respectively.

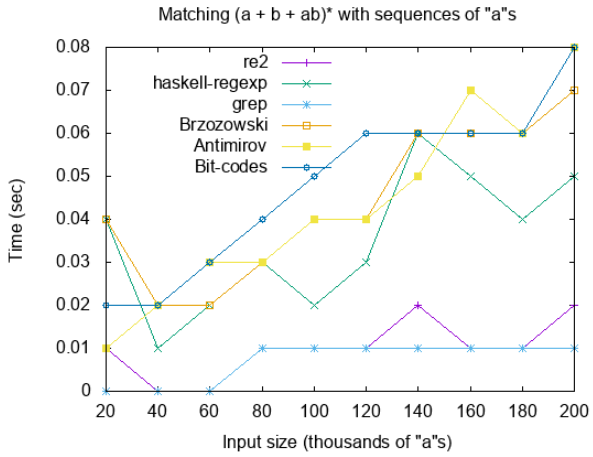

Figure 1: Results of experiment 1.

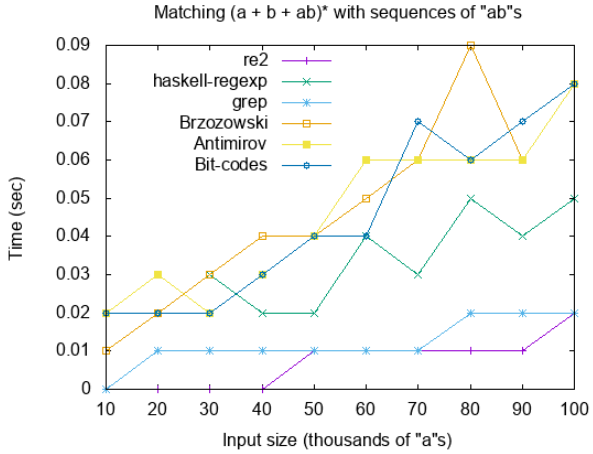

Figure 2: Results of experiment 2.

Our tool behaves poorly when compared with all other options considered. The cause of this inefficiency needs further investigation, since the algorithm formalized uses the POSIX disambiguation strategy, which avoids quotienting the result of derivative operations w.r.t. ACUI axioms as usual Brzozowski derivatives. The main reason behind POSIX and greedy disambiguation strategies in derivative based parsing is to improve efficiency by eliminating simplification steps on derivatives result [33]. We leave the proof that the formalized algorithm indeed produces POSIX parse trees 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 [27]; no formal proof of correctness was presented.

Might et al. [23] 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 [8]. 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 [12]. 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 [28]. This work also does not consider formal proofs of RE parsing.

An algorithm for POSIX RE parsing is described in [33]. 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 NFA represented as a boolean matrix [9]. A matrix library based on some "block" operations [20] is developed and used Agda formalization of NFA-based parsing. 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) [11]. Bernardy et al. describe a formalization of another CFG parsing algorithm in Agda [4]: Valiant's algorithm [34], 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 [17]. The formalized checking procedure verifies if CFG and an automaton match. They proved soundness and completeness of the validator in the Coq proof assistant [5]. Termination of the LR(1) automaton interpreter is ensured by imposing a natural number bound on allowed recursive calls.

Formalization of a parser combinator library was the subject of Danielsson's work [7]. He built a library of parser combinators using coinduction and provides correctness proofs of such combinators.

Almeida et al. [1] describe 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.

Ridge [30] describes a formalization, in the HOL4 theorem prover, of a 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 the Happy parser generator [15].

Ausaf et. al. [3] describe a formalization, in Isabelle/HOL [25], of the POSIX matching algorithm proposed by Sulzmann et.al. [33]. 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.

7 CONCLUSION

We have given a complete formalization of a bit-coded derivative-based parsing for REs in Agda. To the best of our knowledge, this is the first work that presents a complete verification of a bit-code based parsing algorithm and uses it in a tool for RE-based search.

As future work, we intend to continue the development of verigrep by certifying greedy and POSIX disambiguation strategies and finite state machine based algorithms for parsing RE.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This work was conducted during a scholarship supported by PNPD at Universidade Federal de Pelotas, RS - Brazil. Financed by CAPES, Brazilian Federal Agency for Support and Evaluation of Graduate Education within the Ministry of Education of Brazil.

REFERENCES

[1] José Bacelar Almeida, Nelma Moreira, David Pereira, and Simão Melo de Sousa. 2010. 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 (Lecture Notes in Computer Science)*, Michael Domaratzki and Kai Salomaa (Eds.), Vol. 6482. Springer, 59–68. DOI: http://dx.doi.org/10.1007/978-3-642-18098-9_7

[2] Valentin Antimirov. 1996. 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

[3] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. 2016. 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 (Lecture Notes in Computer Science)*, Jasmin Christian Blanchette and Stephan Merz (Eds.), Vol. 9807. Springer, 69–86. DOI: http://dx.doi.org/10.1007/978-3-319-43144-4_5

[4] Jean-Philippe Bernardy and Patrik Jansson. 2016. Certified Context-Free Parsing: A formalisation of Valiant's Algorithm in Agda. *CoRR* abs/1601.07724 (2016). http://arxiv.org/abs/1601.07724

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

[6] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. DOI: http://dx.doi.org/10.1145/321239.321249

[7] Nils Anders Danielsson. 2010. Total Parser Combinators. *SIGPLAN Not.* 45, 9 (Sept. 2010), 285–296. DOI: http://dx.doi.org/10.1145/1932681.1863585

[8] Matthias Felleisen, M.D. Barski Conrad, David Van Horn, and Eight Students of Northeastern University. 2013. *Realm of Racket: Learn to Program, One Game at a Time!* No Starch Press, San Francisco, CA, USA.

[9] Denis Firsov and Tarmo Uustalu. 2013. Certified Parsing of Regular Languages. In *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings (Lecture Notes in Computer Science)*, Georges Gonthier and Michael Norrish (Eds.), Vol. 8307. Springer, 98–113. DOI: http://dx.doi.org/10.1007/978-3-319-03545-1_7

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

[11] Denis Firsov and Tarmo Uustalu. 2014. Certified {CYK} parsing of context-free languages. *Journal of Logical and Algebraic Methods in Programming* 83, 5–6 (2014), 459 – 468. DOI: http://dx.doi.org/10.1016/j.jlamp.2014.09.002 The 24th Nordic Workshop on Programming Theory (NWPT 2012).

[12] Sebastian Fischer, Frank Huch, and Thomas Wilke. 2010. 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, 357–368. DOI: http://dx.doi.org/10.1145/1863543.1863594

[13] Alain Frisch and Luca Cardelli. 2004. Greedy Regular Expression Matching. In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings (Lecture Notes in Computer Science)*, Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella (Eds.), Vol. 3142. Springer, 618–629. DOI: http://dx.doi.org/10.1007/978-3-540-27836-8_53

[14] Grep 2017. GNU Grep home page. https://www.gnu.org/software/grep/. (2017).

[15] Happy 2001. Happy: The parser generator for Haskell. http://www.haskell.org/happy. (2001).

[16] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. 2000. *Introduction to Automata Theory, Languages and Computability* (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[17] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *Proceedings of the 21st European Conference on Programming Languages and Systems (ESOP'12)*. Springer-Verlag, Berlin, Heidelberg, 397–416. DOI: http://dx.doi.org/10.1007/978-3-642-28869-2_20

[18] M. E. Lesk and E. Schmidt. 1990. UNIX Vol. II. W. B. Saunders Company, Philadelphia, PA, USA, Chapter Lex&Mdash;a Lexical Analyzer Generator, 375–387. http://dl.acm.org/citation.cfm?id=107172.107193

[19] Raul Lopes, Rodrigo Ribeiro, and Carlos Camarão. 2015. Certified Derivative-Based Parsing of Regular Expressions. In *Programming Languages — Lecture Notes in Computer Science 9889*. Springer, 95–109.

[20] Hugo Daniel Macedo and José Nuno Oliveira. 2013. Typing linear algebra: A biproduct-oriented approach. *CoRR* abs/1312.4818 (2013). http://arxiv.org/abs/1312.4818

[21] Per Martin-Löf. 1998. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*. Oxford Logic Guides, Vol. 36. Oxford Univ. Press, New York, 127–172.

[22] Conor McBride and James McKinna. 2004. The View from the Left. *J. Funct. Program.* 14, 1 (Jan. 2004), 69–111. DOI: http://dx.doi.org/10.1017/S0956796803004829

[23] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with Derivatives: A Functional Pearl. *SIGPLAN Not.* 46, 9 (Sept. 2011), 189–195. DOI: http://dx.doi.org/10.1145/2034574.2034801

[24] Lasse Nielsen and Fritz Henglein. 2011. *Bit-coded Regular Expression Parsing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 402–413. DOI: http://dx.doi.org/10.1007/978-3-642-21254-3_32

[25] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg.

[26] Ulf Norell. 2009. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09)*. ACM, New York, NY, USA, 1–2. DOI: http://dx.doi.org/10.1145/1481861.1481862

[27] Scott Owens, John Reppy, and Aaron Turon. 2009. Regular-expression Derivatives Re-examined. *J. Funct. Program.* 19, 2 (March 2009), 173–190. DOI: http://dx.doi.org/10.1017/S0956796808007090

[28] re2 2016. Google Regular Expression Library - re2. https://github.com/google/re2. (2016).

[29] Rodrigo Ribeiro, Raul Lopes, and Carlos Camarão. 2017. Certified Derivative Based Parsing of Regular Expressions — On-line repository. https://github.com/rodrigogribeiro/regex. (2017).

[30] Tom Ridge. 2011. 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, 103–118. DOI: http://dx.doi.org/10.1007/978-3-642-25379-9_10

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

[32] Aaron Stump. 2016. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan; Claypool, New York, NY, USA.

[33] Martin Sulzmann and Kenny Zhuo Ming Lu. 2014. POSIX Regular Expression Parsing with Derivatives. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings (Lecture Notes in Computer Science)*, Michael Codish and Eijiro Sumii (Eds.), Vol. 8475. Springer, 203–220. DOI: http://dx.doi.org/10.1007/978-3-319-07151-0_13

[34] Leslie G. Valiant. 1975. General Context-free Recognition in Less Than Cubic Time. *J. Comput. Syst. Sci.* 10, 2 (April 1975), 308–315. DOI: http://dx.doi.org/10.1016/S0022-0000(75)80046-8