Certified Derivative-based Parsing of Regular Expressions

Raul Felipe Pimenta Lopes

A Dissertation submitted for the degree of Master in Computer Science

Departamento de Computação
Universidade Federal de Ouro Preto

March 9, 2018

L881c Lopes, Raul Felipe Pimenta.
Certified derivative-based parsing of regular expressions [manuscrito] / Raul Felipe Pimenta Lopes. - 2018.
50f.: il.: color; grafs.

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro.

Dissertação (Mestrado) - Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação. Programa de Pós-Graduação em Ciência da Computação.
Área de Concentração: Ciência da Computação.

1. Linguagens de domínio específico. 2. Algoritmos de computador. I. Ribeiro, Rodrigo Geraldo. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004.422.4

Catalogação: www.sisbin.ufop.br

Ata da Defesa Pública de Dissertação de Mestrado

Aos 23 dias do mês de fevereiro de 2018, às 15 horas na Sala de Seminários do DECOM no Instituto de Ciências Exatas e Biológicas (ICEB), reuniram-se os membros da banca examinadora composta pelos professores: **Prof. Dr. Rodrigo Geraldo Ribeiro (presidente e orientador), Prof. Dr. Carlos Camarão de Figueiredo, Prof. Dr. José Romildo Malaquias e Prof. Dr. Leonardo Vieira dos Santos Reis**, aprovada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação, a fim de arguirem o mestrando **Raul Felipe Pimenta Lopes**, com o título **"Certified Derivative-based Parsing of Regular Expressions"**. Aberta a sessão pelo presidente, coube ao candidato, na forma regimental, expor o tema de sua dissertação, dentro do tempo regulamentar, sendo em seguida questionado pelos membros da banca examinadora, tendo dado as explicações que foram necessárias.

Recomendações da Banca:

(X) Aprovada sem recomendações

( ) Reprovada

( ) Aprovada com recomendações: ______________________

Banca Examinadora:

[illegible]
Prof. Dr. Rodrigo Geraldo Ribeiro

[illegible]
Prof. Dr. Carlos Camarão de Figueiredo

José Romildo Malaquias
Prof. Dr. José Romildo Malaquias

Leonardo Vieira dos Santos Reis
Prof. Dr. Leonardo Vieira dos Santos Reis

[illegible]
Prof. Dr. Anderson Almeida Ferreira
Coordenador do Programa de Pós-Graduação em Ciência da Computação
DECOM/ICEB/UFOP

Ouro Preto, 23 de fevereiro de 2018.

Contents

Acknowledgements

I would like to thank my advisor, Rodrigo Geraldo Ribeiro for the guidance and advice during the development of this work.

I would like to thank FAPEMIG — Fundação de Amparo à Pesquisa do Estado de Minas Gerais — for the scholarship.

Abstract

Parsing is pervasive in computing and fundamental in several software artifacts. This dissertation reports the first step in our ultimate goal: a formally verified toolset for parsing regular and context free languages based on derivatives. Specifically, we describe the formalization of Brzozowski and Antimirov derivative based algorithms for regular expression parsing, in the dependently typed language Agda. The formalization produces a proof that either an input string matches a given regular expression or that no matching exists. A tool for regular expression based search in the style of the well known GNU Grep has been developed using the certified algorithms. Practical experiments conducted using this tool are reported.

1 Introduction

Parsing is the process of analysing if a string of symbols conforms to a given set of rules. It involves in computer science the formal specification of the rules in a grammar and, also, either the construction of a parsing tree that makes evident which rules have been used to conclude that the string of symbols can be obtained from the grammar rules or, otherwise, an indication of an error that represents the fact that the string of symbols cannot be generated from the grammar rules.

In this work we are interested in the parsing problem for regular languages (RLs) [20], i.e. languages that can be recognized by (non-)deterministic finite automata and equivalent formalisms. Regular expressions (REs) are an algebraic and compact way of specifying RLs that are extensively used in lexical analyser generators [22] and string search utilities [16]. Since such tools are widely used and parsing is pervasive in computing, there is a growing interest on certified parsing algorithms [12, 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.

The use of derivatives for regular expressions were introduced by Brzozowski [9] 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 [30], "derivatives have been lost in the sands of time" until their work on functional encoding of RE derivatives have renewed interest on their use for parsing [27, 14]. In this work, we provide a complete formalization of an algorithm for RE parsing using derivatives [30], and describe a RE based search tool we developed by using the dependently typed language Agda [29].

1.1 Objectives

The main objective of this dissertation is to formalize, in the dependently typed language Agda, algorithms for RE parsing using Brzozowski's derivatives and Antimirov's partial derivatives. [9] [3] The certified algorithms should be used in a RE-based text search tool in the style of the well-known GNU Grep.

1.2 Contributions

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 simplify derivatives using "smart-constructors" [30]. We give formal proofs that smart constructors indeed preserve the language recognized by REs.
- A formalization of Antimirov's partial derivatives and their use to construct a RE parsing algorithm. The main difference between partial derivatives and Brzozowski's is that the former computes a set of REs using set operators instead of "smart-constructors". Producing a set of REs avoids the need of simplification using smart constructors. [3]
- A command-line search tool in the style of GNU-grep [16] that matches regular expressions using the certified algorithms based on Brzozowski and Antimirov derivatives written in Agda.

1.3 Published material

This dissertation builds upon two papers, one published in a peer-reviewed conference and another submitted to a journal.

- The Brzozowski's derivative-based parsing algorithm and its formalization in Idris is described in [23].
- The Antimirov's partial derivative-based algorithm and its formalization in Agda is described in a paper submitted to the Science of Computer Programming journal. This paper is under review.

1.4 Formalization source code

All the source code in this dissertation has been formalized in Agda version 2.5.2 using Standard Library 0.13 but, for space reasons, 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 formalization and instructions on how to build it can be found at the following repository:

`https://www.github.com/raulfpl/regex`

1.5 Dissertation structure

The rest of this dissertation is organized as follows. Chapter 2 provides some necessary background on formal language theory, needed for understanding regular expression parsing; type theory, needed for later chapters on Agda programming language and also presents some related works on verified parsing. Chapter 3 presents a brief introduction to Agda. We describe derivatives for RE and a parsing algorithm in Chapter 4. The main results of this dissertation are reported in Chapter 5, which describes the formalization of the derivative-based RE parsing algorithms. Chapter 6 presents some implementation details of the developed RE parsing tool and experiments performed. Finally, Chapter 7 outlines some possible future works and presents some conclusions obtained from the development of this dissertation.

2 Background

This chapter is concerned with background concepts. We start by reviewing some basic concepts from formal language theory, as found in classic textbooks [20]. Next, we give a concise introduction to type theory, focusing on their relation with intuitionistic logic, known as the Curry-Howard isomorphism [33]. We end this chapter describing some related work on verified parsing algorithms.

A reader familiar with these topics can safely skip this chapter.

2.1 Formal Language Theory

An alphabet is a non-empty finite set of symbols used to build words in a language. Following commom practice, we use the meta-variable Σ to denote an arbitrary alphabet. A string over Σ is a finite sequence of symbols from Σ. We let λ denote the empty string and if x is a string over some alphabet, notation $\mid x \mid$ denote the length of x. We let x^n denote the string formed by n repetitions of x. When $n = 0$, $x^0 = \lambda$. A language over an alphabet Σ is a set of strings over Σ.

Below we present examples of such concepts.

Example 1. Consider the alphabet $\Sigma = \{0, 1\}$. The following are examples of strings over Σ: $\lambda, 0, 1, 00, 111, 0101$. Note that λ is a valid string for any alphabet and, $\mid \lambda \mid = 0$, $\mid 0 \mid = 1$, $\mid 0101 \mid = 4$ and $0^3 = 000$.

Example of languages over $\Sigma = \{0, 1\}$ are $\{0, 11, \lambda\}$ and $\{0^n 1^n \mid n \geqslant 0\}$.

Since languages are sets of strings, we can generate new languages by applying standard set operations, like intersection, union, complement, and so on [20]. In addition to standard set operations, we can build new languages using some operations over strings. Given two languages L_1 and L_2, we define the concatenation, $L_1 L_2$, as:

$$L_1 L_2 = \{xy \mid x \in L_1 \land y \in L_2\}$$

Using concatenation, we can define the iterated concatenation as:

$$\begin{array}{lcl} L^0 & = & \{\lambda\} \\ L^{n+1} & = & L^n L \end{array}$$

Finally, the Kleene closure operator of a language L, $L^\star$, can be defined as:

$$L^\star = \bigcup_{n \in \mathbb{N}} L^n$$

Given an alphabet Σ, $\Sigma^\star$ denote the set of all possible strings formed using symbols from Σ.

A theoretical device (often realized in practical implementations) for formal language processing is the so-called finite state machines. Deterministic finite sate automatas (DFAs) are the type of language recognizers that are capable of accepting Regular Languages.

Definition 1. A deterministic finite automata (DFA) M is a 5-tuple $M = (S, \Sigma, \delta, i, F)$, where:

- S: non empty set of states.
- Σ: input alphabet.
- $\delta : S \times \Sigma \to S$: transition function.
- $i \in S$: initial state.
- $F \subseteq S$: set of final states.

In order to define the set of strings accepted by a DFA, we need to extend its transition function to operate on strings and not only on symbols of its input alphabet as follows:

$$\begin{array}{rcl} \widehat{\delta}(e, \lambda) & = & e \\ \widehat{\delta}(e, ay) & = & \widehat{\delta}(\delta(e, a), y) \end{array}$$

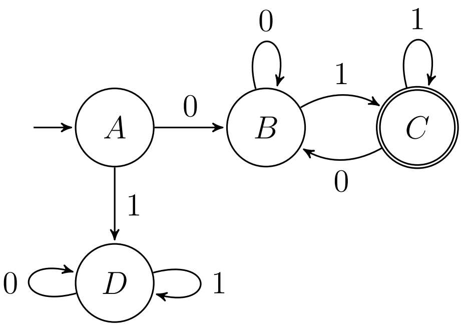
with $e \in S$, $a \in \Sigma$ and $y \in \Sigma^\star$. Using this extended transition function we can define the language accepted by a DFA M as:

$$L(M) = \{w \in \Sigma^\star \mid \widehat{\delta}(i, w) \in F\}$$

Example 2. Consider the following language

$$L = \{w \in \{0, 1\}^\star \mid w \text{ starts with a 0 and ends with a 1}\}$$

A DFA that accepts L is presented below:



From the previous state diagram, the state set S and the final states F are obvious. The following table shows the transition function for this DFA.

δ	0	1
A	B	D
B	B	C
C	B	C
D	D	D

2.1.1 Regular expressions

Regular expressions are an algebraic and widely used formalism for specifying languages in computer science. In this section we will look at the formal syntax and semantics for regular expressions.

Definition 2 (Regular expression syntax)**.** Let Σ be an alphabet. The set of Regular Expressions (REs) over Σ is described by the following grammar:

$$\begin{array}{rcl} e & \rightarrow & \emptyset \\ & \mid & \lambda \\ & \mid & a \\ & \mid & e\,e \\ & \mid & e + e \\ & \mid & e^{\star} \end{array}$$

where $a \in \Sigma$.

We let the meta-variable e denote an arbitrary RE.

A RE describes a set of strings. This is captured by the following definition:

Definition 3 (Regular expression semantics)**.** Let Σ be an alphabet. We define the semantics of a RE over Σ using the following function, $[\![\_]\!] : RE \rightarrow \mathcal{P}(\Sigma^{\star})$ ($\mathcal{P}(x)$ denotes the powerset of a set x):

$$\begin{array}{lcl} [\![\emptyset]\!] & = & \emptyset \\ [\![\lambda]\!] & = & \{\lambda\} \\ [\![a]\!] & = & \{a\} \\ [\![e\,e']\!] & = & [\![e]\!]\,[\![e']\!] \\ [\![e + e']\!] & = & [\![e]\!] \cup [\![e']\!] \\ [\![e^{\star}]\!] & = & ([\![e]\!])^{\star} \end{array}$$

After a precise characterization of RE, we can now use it to define the class of regular languages.

Definition 4 (Regular language)**.** A language $L \subseteq \Sigma^{\star}$ is a regular language (RL) is defined as the fact that there is some RE e such that $L = [\![e]\!]$.

In order to clarify the previous definitions, we present some examples of REs and describe their meaning.

Example 3. Consider $\Sigma = \{0, 1\}$.

- The RE $e = 0^\star 1 0^\star$ denotes the following language
$$L = \{w \in \{0,1\}^\star \mid w \text{ has just one occurrence of } 1\}$$
- The RE $e = (1 + \lambda)0$ denotes the language $L = \{10, 0\}$.
- The RE $e = \emptyset^\star$ denotes the language $L = \{\lambda\}$.
- The RE $e = 0(0 + 1)^\star 1$ denotes the language
$$L = \{w \in \{0,1\}^\star \mid w \text{ starts with a } 0 \text{ and ends with a } 1\}$$

Another way of defining the semantics of REs is as an inductively defined relation between strings and REs. In our Agda formalization we use the inductive semantics instead of the functional one. The inductive semantics for REs is presented in the next definition.

Definition 5 (Inductive semantics for REs)**.** Let Σ be an alphabet, $s \in \Sigma^\star$ and e a RE over Σ. We define that s is in e's language to mean that $s \in e$ can be proved using the following rules.

$$\frac{}{\lambda \in \lambda}\,\textit{Eps} \qquad \frac{a \in \Sigma}{a \in a}\,\textit{Chr} \qquad \frac{s \in e \quad s' \in e'}{ss' \in ee'}\,\textit{Cat}$$

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

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

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

Next, we present a simple example of the inductive RE semantics.

Example 4. The string aab is in the language of RE $(aa+b)^\star$, as the following derivation shows:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \in \Sigma}{a \in a}\,\textit{Chr} \quad \dfrac{a \in \Sigma}{a \in a}\,\textit{Chr}}{aa \in aa}\,\textit{Cat}}{aa \in aa + b}\,\textit{Left} \quad \dfrac{\dfrac{\dfrac{b \in \Sigma}{b \in b}\,\textit{Chr}}{b \in aa + b}\,\textit{Right} \quad \dfrac{}{\lambda \in (aa + b)^\star}\,\textit{StarBase}}{b \in (aa + b)^\star}\,\textit{StarRec}}{aab \in (aa + b)^\star}\,\textit{StarRec}$$

As one would expect, the inductive and functional semantics of REs are equivalent, as shown in the next theorem.

Theorem 1. *For all RE e and strings $s \in \Sigma^\star$, $s \in [\![e]\!]$ if, and only if, $s \in e$.*

Proof. Let e and s be an arbitrary RE and string, respectively.

($\rightarrow$) : Suppose that $s \in [\![e]\!]$. We proceed by induction on the structure of e.

- Case $e = \emptyset$. We have:

$$\begin{aligned} s \in [\![\emptyset]\!] &\leftrightarrow \\ s \in \emptyset &\leftrightarrow \\ &\bot \end{aligned}$$

  which makes the conclusion hold by contradiction.

- Case $e = \lambda$. We have

$$\begin{aligned} s \in [\![\lambda]\!] &\leftrightarrow \\ s \in \lambda & \end{aligned}$$

  Since $e = \lambda$ and $s \in [\![\lambda]\!]$, we have that $s = \lambda$ and the conclusion holds by rule *Eps*.

- Case $e = a$, $a \in \Sigma$. We have:

$$\begin{aligned} s \in [\![a]\!] &\leftrightarrow \\ s \in a & \end{aligned}$$

  Since $e = a$ and $s \in a$, we have that $s = a$ and the conclusion follows by rule *Chr*.

- Case $e = e_1\,e_2$. By the definition of the functional semantics, if $s \in [\![e_1\,e_2]\!]$, then exists $s_1, s_2 \in \Sigma^\star$, such that $s_1 \in [\![e_1]\!]$, $s_2 \in [\![e_2]\!]$ and $s = s_1\,s_2$. By the induction hypothesis, we have that $s_1 \in e_1$ and $s_2 \in e_2$ and the conclusion follows by using rule *Cat*.

- Case $e = e_1 + e_2$. By the definition of the functional semantics, if $s \in [\![e_1 + e_2]\!]$, then $s \in [\![e_1]\!]$ or $s \in [\![e_2]\!]$. Consider the cases:

  * Case $s \in [\![e_1]\!]$: The conclusion follows by the induction hypothesis and rule *Left*.

  * Case $s \in [\![e_2]\!]$: The conclusion follows by the induction hypothesis and rule *Right*.

- Case $e = (e_1)^\star$. Here we proceed by strong induction on the structure of s. Consider the following cases:

  * $s = \lambda$: In this case the conclusion follows by rule *StarBase*.

  * $s \neq \lambda$: Since $s \in ([\![(e_1)]\!])^\star$, by the definition of the Kleene closure, we have that there exists $s_1, s_2 \in \Sigma^\star$ such that $s_1 \in [\![e_1]\!]$, $s_2 \in ([\![e_1]\!])^\star$ and $s = s_1\, s_2$. The conclusion follows by the induction hypothesis and the rule *StarRec*.

$(\leftarrow)$: Suppose that $s \in e$. We proceed by induction on the derivation of $s \in e$ by doing case analysis on the last rule employed to deduce $s \in e$.

- Case *Eps*: We have that $s = \lambda$ and $e = \lambda$. The conclusion follows by the definition of the functional semantics.
- Case *Chr*: We have that $s = a = e$. The conclusion follows by the definition of the functional semantics.
- Case *Cat*: Since the last rule used to deduce $s \in e$ was *Cat*, we have that must exists $s_1, s_2 \in \Sigma^\star$, e_1, e_2 such that $e = e_1\, e_2$, $s = s_1\, s_2$, $s_1 \in e_1$ and $s_2 \in e_2$. By the induction hypothesis, we have that $s_1 \in [\![e_1]\!]$ and $s_2 \in [\![e_2]\!]$. The conclusion follows by the definition of the functional semantics.
- Case *Left*: Since the last rule used to deduce $s \in e$ was *Left*, we have that must exists e_1, e_2 such that $e = e_1 + e_2$ and $s \in e_1$. The conclusion follows by the definition of functional semantics and the indution hypothesis.
- Case *Right*: Since the last rule used to deduce $s \in e$ was *Right*, we have that must exists e_1, e_2 such that $e = e_1 + e_2$ and $s \in e_2$. The conclusion follows by the definition of functional semantics and the indution hypothesis.
- Case *StarBase*: Since the last rule used to deduce $s \in e$ was *StarBase*, we have that $s = \lambda$ and that exists e_1 such that $e = e_1^\star$. The conclusion follows by the definition of functional semantics and the Kleene closure operator.
- Case *StarRec*: Since the last rule used to deduce $s \in e$ was *StarRec*, we have that must exists $s_1, s_2 \in \Sigma^*$, e_1 such that $e = e_1^\star$, $s = s_1\, s_2$, $s_1 \in e_1$ and $s_2 \in (e_1)^\star$. By the induction hypothesis, we have that $s_1 \in [\![e_1]\!]$ and $s_2 \in [\![(e_1)^\star]\!]$ and the conclusion follows from the definition of the functional semantics.

□

2.2 An introduction to type theory

In this section we set the ground for understanding the adequacy of our formalization (or any formalization using a dependently typed language). By adequacy we understand the correspondence between type theoretical objects and other mathematical entities that they represent. Such relation is informally known as the Curry-Howard isomorphism or propositions-as-types correspondence [33].

2.2.1 A whirlwind tour of λ-calculus and types

What is known as the λ-calculus is a collection of formal systems based on the notation invented by Alonzo Church [6]. Church solved the famous *Entscheidungsproblem* (German for decision problem) proposed by Hilbert, in 1928. The challenge consisted in providing an algorithm capable of determining whether or not a given mathematical fact is valid in a given language. Church proved that there is no solution for such problem, that is, it is an undecidable problem.

Church's main objective was to build a formal system for the foundations of mathematics, just like Martin-Löf's type theory, which was to be presented much later, around 1970. Church's original λ-calculus was found to be inconsistent, but later on it was found that there were ways of making it consistent again, with the help of types.

The notion of type is fundamental for this dissertation. This notion arises when we want to combine different terms in a given language. For instance, it makes no sense to try to compute $\int \mathbb{N} dx$. Although syntactically correct, the subterms have different types and, therefore, are not compatible. A type can be seen as a categorization of terms.

For a proper introduction to λ-calculus, the reader is directed to [6, 19]. The goal of this chapter is to provide a minimal knowledge of the λ-calculus, which will allow a better understanding of Martin-Löf's type theory and how logic is encoded in such formalism.

$\lambda-$calculus

Definition 6 (λ-calculus syntax)**.** Let $\mathcal{V}$ be an infinite set of variables and $\mathcal{C}$ an infinite set of constants. The set of terms of the λ-calculus, Λ, is defined by the following grammar, where $v \in \mathcal{V}$, $c \in \mathcal{C}$ and T denotes an arbitrary term.

$$\begin{array}{rcl} T & \rightarrow & c \\ & | & v \\ & | & \lambda v.T \\ & | & T\,T \end{array}$$

Let us adopt some conventions that will be useful throughout this document. Terms will usually be denoted by uppercase letters $M, N, O, P, \ldots$ and variables by lower cases $x, y, z, \ldots$. Following common practice, application is left associative, that is, the term MNO represents $((MN)O)$ whereas abstractions are right associative, so, $\lambda x.\lambda y.K$ represents $(\lambda x.(\lambda y.K))$.

Suppose we have a term $\lambda y.\lambda z.x(yz)$. We say that variables y and z are bounded variables and x is a free variable (as there is no visible abstraction binding it). From now on, we'll use Barendregt's convention for variables and assume that terms do not have any name clashing. In fact, whenever we have two terms M and N that only differ in the naming of their variables, for instance $\lambda x.x$ and $\lambda y.y$, we say that they are α-convertible. Using the concepts of free and bound variables we can define the notion of substitution.

Definition 7 (Substitution)**.** Let M and N be λ-terms where x may have free occurrences in M. The substitution of x by N in M, denoted by $[x \mapsto N]M$ is, informally, the result of replacing every free occurrence of x in M by N. Substitution is defined by induction on M as:

$$\begin{array}{lcll}
[x \mapsto N]x & = & N & \\
[x \mapsto N]y & = & y, x \neq y & \\
[x \mapsto N](M_1\, M_2) & = & ([x \mapsto N]M_1)\, ([x \mapsto N]M_2) & \\
[x \mapsto N](\lambda y.M) & = & \lambda y.[x \mapsto N]M & , x \neq y \\
[x \mapsto N](\lambda x.M) & = & \lambda x.M &
\end{array}$$

Example 5. As an example of substitution, consider the term $\lambda x.\lambda y.x(yz)$ and the substitution $[z \mapsto 1]$. We calculate:

$$\begin{aligned}
[z \mapsto 1]\, \lambda x.\lambda y.x(y\, z) &= \lambda x.\lambda y.[z \mapsto 1]\, x(y\, z) \\
&= \lambda x.\lambda y.([z \mapsto 1]\, x)\, [z \mapsto 1](y\, z) \\
&= \lambda x.\lambda y.([z \mapsto 1]\, x)\, (([z \mapsto 1]\, y)\, ([z \mapsto 1]\, z)) \\
&= \lambda x.\lambda y.x\, (y\, 1)
\end{aligned}$$

β-reduction and confluence

Equipped with both a notion of term and a formal definition of substitution, we can now model the notion of computation. The intuitive meaning is very simple. Imagine a normal function $f(x) = x + 1$ and suppose we want to compute $f(3)$. All we have to do is to substitute x for 3 in the body of $f(x)$, resulting in 3 + 1. This intuition is exactly the notion of computation in the λ-calculus. A term with the form $(\lambda x.M)N$ is called a β-redex and can be reduced to $[x \mapsto N]M$. If a given term has no β-redexes we say that it is a β-normal form.

Definition 8 (β-reduction)**.** Let Λ denote the set of all λ-terms. Suppose that $M, M', N \in \Lambda$ and x is a variable. We let $\to_\beta$ be the following binary relation over Λ defined by induction on M.

$$\frac{}{(\lambda x.M)N \to_\beta [x \mapsto N]M} \qquad \frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

$$\frac{M \to_\beta M'}{M\, N \to_\beta M'\, N} \qquad \frac{M \to_\beta M'}{N\, M \to_\beta N\, M'}$$

We let $\to_\beta^\star$ denote the reflexive-transitive closure of $\to_\beta$. We say that M reduces to N when $M \to_\beta^\star N$.

The relation of β-reduction induces an equivalence relation between terms, named β-equality.

Definition 9 (β-equality)**.** Let M and N be λ-terms. We say that M and N are β equal, denoted by $M =_\beta N$ if $M \to_\beta^\star N$ or $N \to_\beta^\star M$.

Several important results about β-reduction can be found elsewhere [6, 19]. To our objectives the only important property is the unicity of normal forms, i.e., if a term M has a β-normal form, it is unique.

Simply typed λ-calculus

In this section we present the simply typed λ-calculus which is, essentially, the λ-calculus in which all terms have a type.

Definition 10 (Types)**.** Let $\mathcal{T}_C$ be the set of type constants. The syntax of types is defined by the following grammar:

$$\begin{array}{rcl} T & \to & c \\ & | & T \to T \end{array}$$

where $c \in \mathcal{T}_C$.

In any programming task, one is always surrounded by variable declarations. Some languages (the strongly-typed ones) expect some information about the type of such variables. Formally, the information of variables and its types are hold by a *typing context*, which consist of a sequence of pairs formed by a variable and its corresponding type. From now on, we allow ourselves a bit of informality by using set operators on sequences. We let the meta-variable Γ denote arbitrary typing contexts and the symbol $\bullet$ to represent empty typing contexts. As usual, we use the following notations:

$$\Gamma(x) = T \text{ if and only if } x : T \in \Gamma$$
$$\Gamma, x : T = (\Gamma - \{\Gamma(x)\}) \cup \{x : T\}$$

Next, we define the type system of simply typed λ-calculus and the concept of derivability, which will allow us to formally describe the relation between logic and functional programming languages.

Definition 11 (Type system for simply typed λ-calculus and derivability)**.** Let $\varphi : \mathcal{C} \to \mathcal{T}_C$ be a function that assigns to an constant its type. The typing relation for the simply typed λ-calculus, $\Gamma \vdash M : T$, is defined by induction on M as follows:

$$\frac{}{\Gamma \vdash c : \varphi(c)}\ \textit{Const} \qquad\qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textit{Var}$$

$$\frac{\Gamma, x : T' \vdash M : T}{\Gamma \vdash \lambda x : T'.M : T' \to T}\ \textit{Abs} \qquad \frac{\Gamma \vdash M : T' \to T \quad \Gamma \vdash N : T'}{\Gamma \vdash M\ N : T}\ \textit{App}$$

Let Γ be a typing context, M a term and T a type. We say that the sequent $\Gamma \vdash M : T$ is derivable if exists a derivation with $\Gamma \vdash M : T$ in its conclusion.

The intuitive meaning of the previous rules is as follows: Rule (Var) says that a variable x has type T only if $x : T \in \Gamma$. The rule for application says that $M\,N$ is well typed only if it its argument ($N : T'$) has a type that matches $M : T' \to T$ argument type. Finally, we can type an abstraction $\lambda x : T'.M$ with type $T' \to T$ if we type $M : T$ using the additional assumption $x : T'$. Next, we present an example that shows how these rules could be used to type a term.

Example 6. Consider the λ-calculus enriched with integer constants and type $\mathtt{I}$. The term

$$\lambda f : \mathtt{I} \to \mathtt{I}.\lambda x : \mathtt{I}.fx$$

has type $\mathtt{I}$, as deduced by the following derivation, using the empty typing context.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(f) = \mathtt{I} \to \mathtt{I}}{\{f : \mathtt{I} \to \mathtt{I}, x : \mathtt{I}\} \vdash f : \mathtt{I} \to \mathtt{I}}\,{\scriptstyle Var} \quad \dfrac{\Gamma(x) = \mathtt{I}}{\{f : \mathtt{I} \to \mathtt{I}, x : \mathtt{I}\} \vdash x : \mathtt{I}}\,{\scriptstyle Var}}{\{f : \mathtt{I} \to \mathtt{I}, x : \mathtt{I}\} \vdash fx : \mathtt{I}}\,{\scriptstyle App}}{\{f : \mathtt{I} \to \mathtt{I}\} \vdash \lambda x : \mathtt{I}.fx : \mathtt{I}}\,{\scriptstyle Abs}}{\bullet \vdash \lambda f : \mathtt{I} \to \mathtt{I}.\lambda x : \mathtt{I}.fx : \mathtt{I}}\,{\scriptstyle Abs}$$

In this case, we can say that $\bullet \vdash \lambda f : \mathtt{I} \to \mathtt{I}.\lambda x : \mathtt{I}.fx : \mathtt{I}$ is derivable.

The Curry-Howard Isomorphism

On one hand we have the models of computation, on the other hand we have the proof systems. At a first glance, they look like very different formalisms, but, after a closer look, they turned out to be structurally the same. Let M be a term and Γ a context such that $\Gamma \vdash M : T$ is derivable, for some T. We can look at T as a propositional formula and to M as a proof of such formula[1], in natural deduction [19]. In order to show such similarity between these formalisms (type theory and logic), let's put the rules for each one side-by-side.

Natural deduction	Type derivation
$\dfrac{T \in \Gamma}{\Gamma \vdash T}\,{\scriptstyle Id}$	$\dfrac{\Gamma(x) = T}{\Gamma \vdash x : T}\,{\scriptstyle Var}$
$\dfrac{\Gamma, T' \vdash T}{\Gamma \vdash T' \supset T}\,{\scriptstyle \supset_I}$	$\dfrac{\Gamma, x : T' \vdash M : T}{\Gamma \vdash \lambda x.M : T' \to T}\,{\scriptstyle Abs}$
$\dfrac{\Gamma \vdash T' \supset T \quad \Gamma \vdash T'}{\Gamma \vdash T}\,{\scriptstyle \supset_E}$	$\dfrac{\Gamma \vdash M : T' \to T \quad \Gamma \vdash N : T'}{\Gamma \vdash M\,N : T}\,{\scriptstyle App}$

This seemingly shallow equivalence is a remarkable result in Computer Science, discovered by Curry and Howard [19]. This was the starting point for the first proof checkers,

[1]Note that, implication, here denoted by $\supset$, together with variables and falsity, $\bot$, forms a complete set of connectives, enough to express any formula in propositional logic.

since checking a proof is the same as typing a λ-term. If the term is typeable, then the proof is valid. This far we have only presented the simpler version of this connection. Another layer will be built on top of it and add all ingredients for working over first-order logic, in the next section. Behind the curtains, all Agda does is type-checking terms. The understanding of this connection is of major importance for writing proofs and programs in Agda (or any other proof-assistant based on the Curry-Howard isomorphism, for that mater).

Martin Löf type theory

Type theory was originally developed with the goal of offering a clarification, or basis, for constructive Mathematics. However, unlike most other formalizations of Mathematics, it is not based on first order logic. Therefore, we need to introduce the symbols and rules we'll use before presenting the theory itself. The core of this interpretation of proofs as programs is the Curry-Howard isomorphism, already explained in Section 2.2.1.

Martin-Löf's theory of types [25] is an extension of regular type theory. This extended interpretation includes universal and existential quantification. A proposition is interpreted as a set whose elements are proofs of such proposition. Therefore, any true proposition is a non-empty set and any false proposition is an empty set, meaning that there is no proof for such proposition. Apart from sets as propositions, we can look at sets from a specification angle, and this is the most interesting view for programming. A given element of a set A can be viewed as: a proof for proposition A; a program satisfying the specification A; or even a solution to problem A.

Constructive Mathematics

The line between Computer Science and Constructive Mathematics is blurry. The primitive object is the notion of a function from a set A to a set B. Such function can be viewed as a program that, when applied to an element $a \in A$ will construct an element $b \in B$. This means that every function we use in Constructive Mathematics is computable.

Using constructivism to prove things is related to building a computer program. For example, to prove a proposition $\forall x_1, x_2 \in A.\exists y \in B.Q(x_1, x_2, y)$ for a given predicate Q is equivalent to giving a function that when applied to two elements $a_1, a_2 \in A$ will give an element $b \in B$ such that $Q(a_1, a_2, b)$ holds.

Propositions as sets

In Classical Mathematics, a proposition is thought of as being either true or false, and it's proof it's a separate matter. On a different angle, a proposition is constructively true if we have a method for proving it. A classical example is the law of excluded middle, $A \vee \neg A$, which is trivially true since A can only be true or false. Constructively, though, a method for proving a disjunction must prove that one of the disjuncts holds. Since we cannot prove an arbitrary proposition A, we have no proof for $A \vee \neg A$.

Therefore, we have that the constructive explanation of propositions is built in terms of proofs, and not an independent mathematical object. The interpretation we are going to present here is due to Heyting at [18].

Falsity: $\perp$, is identified as the empty set, $\emptyset$. That is, a set with no elements or a propositional without proof.

Conjunction: $A \wedge B$ is identified with the cartesian product $A \times B$. A proof of $A \wedge B$ is a pair formed by a proof of A and a proof of B.

Disjuntion: $A \vee B$ is identified with the disjoint union $A \uplus B$. A proof $A \vee B$ is either a proof of A or a proof of B.

Implication: $A \supset B$ is the set of functions from A to B, denoted as B^A. That is, a proof of $A \supset B$ is a method that build a proof of B from a proof of A.

Negation: $\neg A$ is seen as functions that from a proof of A, produces an absurd, i.e. $\neg A \equiv A \supset \perp$, as in propositional logic.

We have defined propositional logic using sets (types) that are available in almost every functional programming language. Quantifications, though, require operations defined over a family of sets, possibly depending on a given value. The intuitionist explanation of the existential quantifier is as follows:

Existential quantifier: $\exists a \in A.P(a)$ is a pair whose first component is one element $x \in A$ and whose second component is a proof of $P(x)$. More generally, we can identify it with the disjoint union of a family of sets, denoted by $\Sigma(x \in A, B(x))$, or just $\Sigma(A, B)$. The elements of $\Sigma(A, B)$ are of the form (a, b) where $a \in A$ and $b \in P(a)$.

Universal quantifier: $\forall a \in A.P(a)$, is a function that gives a proof of $P(a)$ for each $a \in A$ given as input. The correspondent set is the cartesian product of a family of sets $\Pi(x \in A, B(x))$. The elements of such set are the aforementioned functions. The same notation simplification takes place here, and we denote it by $\Pi(A, B)$. The elements of such set are of the form $\lambda x.b(x)$ where $b(x) \in B(x)$ for $x \in A$.

2.2.2 Epilogue

At this point we have briefly discussed the Curry-Howard isomorphism in both the simply-typed and the dependently typed flavor. We have presented (the very surface of) the theory of types and which kind of logical judgments we can handle with it. But how does all this connect to Agda and verification in general?

Software Verification with a functional language is somewhat different from doing the same with an imperative language. If one is using C, for example, one would write the program and annotate it with logical expressions. They are twofold. We can use pre-conditions, post-conditions and invariants to deductively verify a program (these are in fact a variation of Hoare's logic). Or, we can use Software Model Checking, proving that for some bound of traces k, our formulas are satisfied.

When considering functional languages, our code is correct by construction. The type-system provides the annotation language and you can construct a program that either type-checks, therefore respects its specification, or does not pass through the compiler. Remember that Agda's type system is the intensional variant of Martin-Löf's theory of types, which was described succinctly in this chapter.

Readers interested in a more recent, revised and detailed presentation of Martin-Löf's type theory should consult [36].

2.3 Related works

Parsing with derivatives Recently, derivative-based parsing has received a lot of attention. Owens et al [30]. 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; no formal proof of correctness was presented.

Might et al. [27] 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 is described by using the Racket programming language [11]. 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 [14]. 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 [31]. This work also does not consider formal proofs of RE parsing.

An algorithm for POSIX RE parsing is described in [35]. 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. [35] 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 [12]. A matrix library based on some "block" operations [24] is developed and used in the Agda formalization of NFA-based parsing [29]. Compared to our work, a NFA-based formalization requires much more infrastructure (such as a Matrix library). No experiments with the certified algorithm were reported.

Firsov describes an Agda formalization of a parsing algorithm that deals with any CFG (CYK algorithm) [13]. Bernardy et al. describe a formalization of another CFG parsing algorithm in Agda [7]: Valiant's algorithm [37], which reduces a CFG parsing problem to boolean matrix multiplication. In both works, no experiment with formalized parsing algorithms were reported.

A certified LR(1) CFG validator is described in [21]. The formalized checking procedure verifies if a CFG and an automaton match. They proved soundness and completeness of the validator in the Coq proof assistant [8]. 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 [10]. He built a library of parser combinators using coinduction and provide correctness proofs of such combinators.

Almeida et al. [2] describe a Coq formalization of partial derivatives and its equivalence with automata. Partial derivatives were introduced by Antimirov [4] as an alternative to Brzozowski derivatives, since it avoids quotient resulting REs with respect to ACUI axioms(Associativity, Commutativity and Idempotence with Unit elements axioms for REs[9]). Almeida et al. motivation is to use such formalization as a basis for a decision procedure for RE equivalence.

Ridge [32] 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 show that parsers built using his generator are faster than those created by the Happy parser generator [17].

Ausaf et al. [5] describe a formalization, in Isabelle/HOL [28], of the POSIX matching algorithm proposed by Sulzmann et al. [35]. 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 have been reported.

3 The Agda programming language

Agda is an implementation of Martin-Löf's type theory [25], extended with records and modules. Thanks to the Curry–Howard isomorphism, it is both a functional programming language and a proof assistant for intuitionistic logic.

This chapter aims to be a short introduction to Agda. We assume that the reader knows Haskell, which will allow us to avoid explaining some Agda syntatic constructs that are inspired by Haskell's.

3.1 Data types in Agda

The most basic example in dependently typed languages, like Agda, involves functions around natural numbers in Peano notation.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

In Agda, a single colon means "is of type". So, in the previous source code snipet, we have that zero has type ℕ and suc has type ℕ → ℕ, i.e. it takes a natural number argument and returns a new natural number. Also, the type ℕ has a type, which is Set. In Agda, Set is the type of types: types are first class values.

As in Haskell, we can build functions by pattern-matching. Our first example is the addition of natural numbers:

```
_+_ : ℕ → ℕ → ℕ
zero + m = m
(suc n) + m = suc (n + m)
```

Notice how we write _+_ for the name of the function, then later drop the underscores. This notation is referred to as mixfix — in Agda we are allowed to define operators using underscores to denote where they expect arguments. Other than that, addition is fairly straightforward, using the inductive style of programming we all know and love.

We will now look at the definition of lists in Agda. This is already starting to look slightly different to the corresponding Haskell implementation.

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
```

The first thing to note is that we are allowed to use Unicode symbols for function and constructor names — the combination of mixfix and Unicode makes Agda very liberal in what is accepted as an identifier. The next thing to note is that the List data type is parameterised by an argument, A, of type Set. Recall that Set is the type of types, so List is parameterised by a type for the values it should contain.

A useful function on lists is head, which returns the first element of an input list. A Haskell programmer would write something like:

```
head : List A → A
head (x :: xs) = x
```

Unfortunately, this implementation of function head isn't accepted by Agda's compiler, since it isn't *total* and do not declare all of its parameters. Next, we present a new definition of head that defines all of its parameters.

```
head : (A : Set) → List A → A
head A (x :: xs) = x
```

In Agda, types are first class citizens, they can passed as arguments to functions and returned as results. However, this definition is still not accepted by Agda but it has introduced another inconvenience: Every call of head function must provide, as argument, the type of the input list elements like: head ℕ (zero :: suc zero :: [ ]). To avoid such spurious type arguments, Agda allow us to declare them as *implicit arguments*, i.e. arguments that can be infered by the compiler using the expression surrounding context. Implicit arguments are declared using curly braces. The next definition of head defines A : Set as an implicit argument.

```
head : {A : Set} → List A → A
head (x :: xs) = x
```

Again, this definition of head is still not accepted by Agda because it is not *total*. We say that a function is total if it terminates and is defined for all possible inputs. In Agda, all functions are required to be total. Termination is checked by making sure that recursive calls are always done on structurally smaller arguments — as is the case in the addition example. Furthermore, a function is considered to be defined on all inputs when the patterns it matches on cover all the possibilities of its input type. What we mean by this is that an alternative should be given for each possible constructor — something which is violated by the head attempt: it is missing a case for the empty list. This is something Haskell does not care about; it simply smirks and throws an exception if we try to normalise the expression head [ ]. A solution to make head total is to use the type Maybe, a standard solution in Haskell.

```
head : {A : Set} → List A → Maybe A
head [ ] = nothing
head (x :: xs) = just x
```

Using type Maybe is something of an annoyance; it would be preferable to guarantee that the empty list is not valid input to the head function. This is where a dependently typed language really comes into its own. We will now move on to the *hello world* example of dependent types, the vector, or length indexed lists.

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

The reader must have noticed that: 1 - Agda allows the overloading of data type constructors, since Vec and List constructors are denoted by the same identifiers; and 2 - in Vec type definition, we have an argument both to the left and the right of the colon in the type signature of the data definition. Left-hand arguments are called parameters, and scope over all the constructors. Right-hand arguments are called indices, and only scope over single constructors, and as such need to be introduced per constructor. The _::_ constructor has a size parameter, which is an example of such an index. This dependent type has the advantage that we can distinguish vectors of different sizes by their type, at compile time, without knowing their value.

Using type Vec we can write a safe vector head function without resorting to type Maybe:

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

Note that only vectors with a value n such that suc n is the length of the vector, are valid inputs. This way, we guarantee that empty vectors cannot be passed as arguments to head. Agda is also convinced that this function is total, since the empty list cannot be given the type Vec A (suc n), for any value of n.

Agda data types can be used to define inductive propositions. As example, consider the following judgement to construct eveness proofs for natural numbers:

$$\frac{}{\text{Even } \; zero}\; \textit{EvZero} \qquad \frac{\text{Even}\, n}{\text{Even } \; suc\,(suc\, n)}\; \textit{EvSuc}$$

The previous rules define the necessary conditions for a natural number n be even:

- $n = zero$, or
- $n = suc(suc\, m)$, where m is an even number.

We can achieve the same meaning using an inductive type in Agda. Each rule of the previous judgment will become a constructor in its corresponding type. Below, we present type Even that can be used to build eveness proofs for natural numbers in Peano notation.

```
data Even : ℕ → Set where
  zero : Even zero
  suc : ∀ {n} → Even n → Even (suc (suc n))
```

Next we present example proofs for eveness both in natural deduction style and Agda code.

Example 7. Using the previous eveness proof rules we can show that Even 6, as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\text{Even } 0}\,EvZero}{\text{Even } 2}\,EvSuc}{\text{Even } 4}\,EvSuc}{\text{Even } 6}\,EvSuc$$

Using the Even Agda type, the same proof can be represented as the following term:

```
ev6 : Even 6
ev6 : suc (suc (suc zero))
```

where suc and zero are the constructors of Even that denote the rules $EvSuc$ and $EvZero$, respectively.

Another useful type is the decidable proposition type, which 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 $P \rightarrow \bot$, 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). Note that the decidable type Dec P corresponds to a constructive interpretation of excluded middle, which obviously holds only for propositions that are decidable.

3.2 Pattern matching in Agda

So far we have seen the very basics of Agda. A few aspects deserve more attention, though, one prime example being Agda's pattern matching facilities. One of Haskell's best things is the ability to do pattern matching using a very clean syntax. Agda shares this idiomatic programming style, but has a much more powerful version of pattern matching, namely dependent pattern matching.

3.2.1 The with construct

Dependently typed pattern matching is built by using the so-called **with** construct, that allows for matching intermediate values [26]. 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 $n = suc\ (k + k)$, 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 (suc (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
```

3.2.2 Absurd patterns

Since dependent types allows the reference of values in types, some combinations of types are simply not allowed, or *absurd* in Agda terminology. As an example, consider the task of developing a safe list indexing function. In order to exclude invalid indexes, we must relate the index type with the size present in Vec A n type. The type *Fin* n denotes a type inhabited by exactly $n + 1$ values.

```
data Fin : ℕ → Set where
  zero : ∀ {n} → Fin (suc n)
  suc  : ∀ {n} → Fin n → Fin (suc n)
```

From Fin data type definition, we can see that there's no possible value of type Fin zero, which means that Fin zero is an unprovable (false) proposition. Such impossible situations are represented in Agda patterns by *absurd patterns*, written as (). As an example, consider the safe list indexing function:

```
_!_ : ∀ {A n} → Vec A n → Fin n → A
[ ] ! ()
(x :: xs) ! zero = x
(x :: xs) ! (suc ix) = xs ! ix
```

In the first equation, we matched on the empty Vec which forces the index n to be equal to zero. In such situation, we have that the second parameter must be a value of type Fin zero, which is not possible, since it is not possible, from Fin definition, to build any value of type Fin zero. Whenever we have a pattern whose type is an empty type, we can use the *absurd* pattern to omit the right-hand side of this equation.

Absurd patterns can be used to define a form of reasoning common in formal logic known as *ex falsum quodlibet*[1], which allows us to deduce anything from a false proposition:

```
data ⊥ : Set where

exFalsum : ∀ {A : Set} → ⊥ → A
exFalsum ()
```

In previous source code snipet, we have a definition of type ⊥, with no constructors, which can be understood as the false proposition. Function exFalsum denotes the ex falsum quodlibet principle, in Agda code. It will return a value of A from a proof of ⊥.

3.2.3 Inferrable patterns

Another important pattern-matching feature of Agda is the *dotted-patterns*. Because pattern matching is dependent, information about certain arguments can often be inferred from others. Using a dotted pattern means certain parameters are inferable or equal to others.

Consider the type of the so-called propositional equality:

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

The type _≡_ only contains values constructed using refl (which stands for reflexivity), and refl can only be used when the arguments to _≡_ are β-equal. This is because the same x is used as both first and second argument to _≡_.

We might use the equality type as follows; we are writing a function which is only defined on equal naturals. Here we pattern match on whether some naturals are equal, and if so, we can use this information on the left-hand side of the equation too. Note how repeated variables on the left-hand side are allowed, if their value is inferable.

```
weird : (n m : ℕ) → n ≡ m → List ℕ
weird .m m refl = []
```

3.3 Concluding remarks

In this chapter, we presented a small introduction to Agda by telling a tiny bit of what the language is capable of, both in its programming side and its proof assistant side. More information on Agda can be found elsewhere [34] [1].

[1]Latin for “from false, anything follows”.

4 Derivatives for regular expressions

In this chapter, we present both Brzozowski's derivatives and Antimirov's partial derivatives. After defining these operations over REs, we show how they can be used to develop a parsing algorithm for REs.

4.1 Derivatives, in general

Given a symbol $a \in \Sigma$ and a language over Σ, $L \subseteq \Sigma^\star$, the derivative of L with respect to a, L_a, is defined as:

$$L_a = \{w \mid aw \in L\}$$

We can extend the quotient to operate over words straightforwardly (s denotes an arbitrary string from the input alphabet Σ):

$$L_s = \{w \mid sw \in L\}$$

Also, RLs are closed with respect to the quotient operation, as proved by the next theorem.

Theorem 2. *If L is a regular language over Σ and $a \in \Sigma$ then the quotient of L with respect to a is a regular language.*

Proof. Suppose that L is a regular language. Then, exists a DFA $M = (E, \Sigma, \delta, i, F)$ such that $L(M) = L$. We need to show that the DFA $M' = (E, \Sigma, \delta, \delta(i, a), F)$ accepts L_a, i.e. we need to prove that:

$$aw \in L \rightarrow \widehat{\delta}(i, aw) \in F \leftrightarrow \widehat{\delta}(\delta(i, a), w) \in F$$

which is immediate from the definition of $\widehat{\delta}$:

$$\begin{array}{lcl} \widehat{\delta}(e, \lambda) & = & e \\ \widehat{\delta}(e, ay) & = & \widehat{\delta}(\delta(e, a), y) \end{array}$$

□

The previous definitions give an element (string) based definition for the quotient operation. In the next sections we present symbolic procedures that produce REs for the quotient operation.

4.2 Brzozowski's derivatives

The main insight of Brzozowski's derivative definition is to build a symbolic method that computes, from a RE and a symbol, a new RE that denotes the quotient of the input RE with respect to the input symbol [9].

In order to define the derivative of a RE e, we need an auxiliary function, named nullability test (denoted by ν), which returns λ, when $\lambda \in [\![e]\!]$ and $\emptyset$, otherwise.

$$\begin{array}{lcl}
\nu(\emptyset) & = & \emptyset \\
\nu(\lambda) & = & \lambda \\
\nu(a) & = & \emptyset \\
\nu(e\,e') & = & \left\{\begin{array}{ll} \lambda & \text{when } \nu(e) = \nu(e') = \lambda \\ \emptyset & \text{otherwise.} \end{array}\right. \\
\nu(e + e') & = & \left\{\begin{array}{ll} \lambda & \text{when } \nu(e) = \lambda \text{ or } \nu(e') = \lambda \\ \emptyset & \text{otherwise.} \end{array}\right. \\
\nu(e^\star) & = & \lambda
\end{array}$$

From ν's definition, we can easily prove the following.

Lemma 1. *For all RE e, $\nu(e) = \lambda$ if, and only if, $\lambda \in e$.*

Proof. By induction on e's structure. In both directions, the cases for $e = \emptyset$, $e = \lambda$, $e = a$ or $e = e_1^\star$ are immediate. For concatenation and union, the result follows by the induction hypothesis. □

Using the nullability test, we can define the function to compute the derivative of an input RE with respect to a symbol $a \in \Sigma$ as follows:

$$\begin{array}{lcl}
\delta_a(\emptyset) & = & \emptyset \\
\delta_a(\lambda) & = & \emptyset \\
\delta_a(b) & = & \left\{\begin{array}{ll} \lambda & \text{if } b = a \\ \emptyset & \text{otherwise} \end{array}\right. \\
\delta_a(e\,e') & = & \delta_a(e)\,e' + \nu(e)\,\delta_a(e') \\
\delta_a(e + e') & = & \delta_a(e) + \delta_a(e') \\
\delta_a(e^\star) & = & \delta_a(e)\,e^\star
\end{array}$$

From δ's definition, we can see that for any e and a, $\delta_a(e)$ is a RE. Next, consider the following example that illustrates how to use the previous definition to calculate the derivative of a RE.

Example 8. Consider the RE $e = (aa + b)^\star$. The derivative of e with respect to a is $a(aa + b)^\star$, as demonstrated below:

$$\begin{aligned}
\delta_a((aa + b)^\star) &= \delta_a(aa + b)\,(aa + b)^\star \\
&= [\delta_a(aa) + \delta_a(b)]\;(aa + b)^\star \\
&= [a + \emptyset]\;(aa + b)^\star \\
&= a(aa + b)^\star
\end{aligned}$$

Derivatives can be extended to strings in the standard way:

$$\begin{array}{rcl}\delta^\star_\lambda(e) & = & e\\ \delta^\star_{ay}(e) & = & \delta^\star_y(\delta_a(e))\end{array}$$

As expected, the derivative operation is sound and complete with respect to the RE semantics.

Theorem 3. *For all RE e, $a \in \Sigma$ and $w \in \Sigma^\star$, $aw \in e$ if, and only if, $w \in \delta_a(e)$.*

Proof. Consider arbitrary e, $a \in \Sigma$ and $w \in \Sigma^\star$. Both sides proceed by induction on the inductive RE semantics, using similarity to quotient derivative results. □

In practical terms, applying the derivative operation on REs can result in expressions which can be simplified, while denoting the same language. Owens et al. [30] define a similarity relation, which can be understood as an approximation of the standard notion of RE equivalence.

We say that e is equivalent to e', $e \equiv e'$, if they denote the same language, i.e. $[\![e]\!] = [\![e']\!]$. Let $e \approx e'$ be the least relation on RE's with satisfies the following equations:

$$\begin{array}{rclrcl} e + e & \approx & e & e + e' & \approx & e' + e\\ e + (e' + e'') & \approx & (e + e') + e'' & \emptyset + e & \approx & e\\ e\,(e'\,e'') & \approx & (e\,e')\,e'' & \emptyset\, e & \approx & \emptyset\\ e\,\emptyset & \approx & \emptyset & \lambda\, e & \approx & e\\ e\,\lambda & \approx & e & (e^\star)^\star & \approx & e^\star\\ \lambda^\star & \approx & \lambda & \emptyset^\star & \approx & \lambda \end{array}$$

When $e \approx e'$ holds we say that e and e' are *similar* [30]. Evidently, we have that similarity is compatible with the standard notion of RE equivalence.

Theorem 4. *For all REs e and e', if $e \approx e'$ then $e \equiv e'$.*

Proof. Induction on the rules defining $e \approx e'$. Base cases are immediate using the definition of RE semantics and inductive ones are immediate from induction hypothesis. □

In our Agda formalization, we use "smart-constructors" [30] to quotient similar REs using similarity. The intent of such simplification is just for performance reasons and do not affect the termination or correctness of the parsing algorithm. We leave a detailed discussion on smart-constructors to Section 5.2.

4.2.1 RE parsing using Brzozowski's derivatives

Note that, $s \in e$ if, and only if, $\lambda \in \delta^\star_s(e)$, which is true whenever $\nu(\delta^\star_s(e)) = \lambda$. Owens et al. [30] define a relation between strings and RE, called the *matching* relation, as:

$$\begin{array}{rcl} e \sim \lambda & \Leftrightarrow & \nu(e)\\ e \sim as & \Leftrightarrow & \delta_a(e) \sim s \end{array}$$

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

Theorem 5. *For all $s \in \Sigma^\star$ and REs e, $s \in e$ if, and only if, $e \sim s$.*

Proof. Straightforward induction on e in both directions. □

Example 9. Suppose $e = (aa + b)^\star$ and the string aab. We can show that $aab \in e$ using the previously defined matching relation as follows:

$$\begin{aligned}
(aa+b)^\star \sim aab &\Leftrightarrow \delta_a(aa+b)(aa+b)^\star \sim ab \\
&\Leftrightarrow a(aa+b)^\star \sim ab \\
&\Leftrightarrow \delta_a(a(aa+b)^\star) \sim b \\
&\Leftrightarrow (aa+b)^\star \sim b \\
&\Leftrightarrow \delta_b(aa+b)(aa+b)^\star \sim \lambda \\
&\Leftrightarrow (aa+b)^\star \sim \lambda \\
&\Leftrightarrow \nu((aa+b)^\star) = \lambda \text{ which is true.}
\end{aligned}$$

The same idea can be used to show that a string isn't in the language of some RE. Now consider again the RE $e = (aa + b)^\star$ and the string abb:

$$\begin{aligned}
(aa+b)^\star \sim abb &\Leftrightarrow \delta_a(aa+b)(aa+b)^\star \sim bb \\
&\Leftrightarrow a(aa+b)^\star \sim bb \\
&\Leftrightarrow \delta_b(a(aa+b)^\star) \sim b \\
&\Leftrightarrow \delta_b(a)(aa+b)^\star \sim b \\
&\Leftrightarrow \emptyset\,(aa+b)^\star \sim b \\
&\Leftrightarrow \emptyset \sim b \\
&\Leftrightarrow \delta_b(\emptyset) \sim \lambda \\
&\Leftrightarrow \nu(\emptyset) = \lambda \text{ which is false.}
\end{aligned}$$

4.3 Antimirov's partial derivatives

While Brzozowski's derivatives can be understood as a method for computing deterministic automata from REs, Antimirov's partial derivatives allows the construction of non-deterministic automata from a RE [4]. The main insight of partial derivatives is to build a set of REs which, collectively, accept the same language as Brzozowski's derivatives.

Partial derivatives avoid the need of quotient similar REs by using standard set operations which, by construction, guarantee that some equations of similarity hold. In the original paper [4], Antimirov uses only smart-constructors for simplifying concatenation.

Below, we present the definition of partial derivative computation, where we consider that $\odot$ has greater precedence than $\cup$.

$$\begin{array}{lcl}
\nabla_a(\emptyset) & = & \emptyset \\
\nabla_a(\lambda) & = & \emptyset \\
\nabla_a(b) & = & \left\{ \begin{array}{ll} \{\lambda\} & \text{if } b = a \\ \emptyset & \text{otherwise} \end{array} \right. \\
\nabla_a(e\,e') & = & \left\{ \begin{array}{ll} \nabla_a(e) \odot e' \cup \nabla_a(e') & \text{if } \nu(e) = \lambda \\ \nabla_a(e) \odot e' & \text{otherwise} \end{array} \right. \\
\nabla_a(e + e') & = & \nabla_a(e) \cup \nabla_a(e') \\
\nabla_a(e^\star) & = & \nabla_a(e) \odot e^\star
\end{array}$$

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

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

Next, we present an example of how to use the previous definition to compute the set of partial derivatives of an input RE.

Example 10. Consider RE $e = (aa+b)^\star$. The set of partial derivatives of e with respect to a is $\{a(aa+b)^\star\}$ as demonstrated below:

$$\begin{array}{rl}
\nabla_a((aa+b)^\star) & = \nabla_a(aa+b) \odot (aa+b)^\star \\
& = [\nabla_a(aa) \cup \nabla_a(b)] \odot (aa+b)^\star \\
& = \{[\nabla_a(a) \odot a \cup \emptyset]\} \odot (aa+b)^\star \\
& = \{[\{\lambda\} \odot a \cup \emptyset]\} \odot (aa+b)^\star \\
& = \{a\} \odot (aa+b)^\star \\
& = \{a(aa+b)^\star\}
\end{array}$$

Next theorem relates Antimirov's partial derivatives with RE semantics.

Theorem 6. *For all REs e, $a \in \Sigma$ and $w \in \Sigma^\star$, $aw \in e$ if, and only if, there exists $e' \in \nabla_a(e, w)$.*

Proof. By induction on the definition of RE semantics. □

We can also lift partial derivatives to strings as follows:

$$\begin{array}{lcl}
\nabla^\star(S, \lambda) & = & S \\
\nabla^\star(S, aw) & = & \bigcup_{e \in S} \nabla^\star(\nabla_a(e), w)
\end{array}$$

A string w matches a RE e if $\exists e'.e' \in \nabla^\star(\{e\}, w) \wedge \nu(e') = \lambda$. Next example shows how partial derivatives can be used for parsing REs.

Example 11. Suppose $e = (aa+b)^\star$ and the string aab. We can show that $aab \in e$ using the previously defined matching relation as follows:

$$\begin{array}{rcl}
\nabla^{\star}(\{(aa+b)^{\star}\}, aab) & = & \nabla^{\star}(\{a(aa+b)^{\star}\}, ab) \\
& = & \nabla^{\star}(\{(aa+b)^{\star}\}, b) \\
& = & \nabla^{\star}(\{\nabla_b(aa+b) \odot (aa+b)^{\star}\}, \lambda) \\
& = & \nabla^{\star}([\nabla_b(aa) \cup \nabla_b(b)] \odot (aa+b)^{\star}, \lambda) \\
& = & \nabla^{\star}((aa+b)^{\star}, \lambda) \\
& = & \{(aa+b)^{\star}\}
\end{array}$$

and, since $\exists e'.e' \in \nabla^{\star}(\{(aa+b)^{\star}\}, w) \wedge \nu(e') = \lambda$, we can say that $aab \in (aa+b)^{\star}$.

4.4 Concluding remarks

In this chapter, we provide definitions of derivatives and partial derivatives as proposed by Brzozowski and Antimirov, respectively. We have provided some proof sketches of their main properties and presented some examples.

We want to emphasize that, for our purposes, understanding RE parsing as a matching relation isn't adequate because RE-based text search tools, like GNU Grep, shows every matching prefix and substring of a RE for a given input. Since our interest is determining which prefixes and substrings of the input string match a given RE, in the next chapter we adapt the matching algorithm in order to compute prefixes and substrings of a given input, and present correctness proofs.

5 Agda formalization

In this chapter we describe the Agda formalization of the parsing algorithms described briefly in Chapter 4. We start by defining RE syntax as an Agda data type and how to encode the inductive RE semantics as an inductive proposition (Section 5.1). In Section 5.2 we define smart-constructors, used to simplify REs during derivative computation. The functions for computing derivatives, partial derivatives and expressing and proving their properties are presented in Sections 5.3 and 5.4, respectively. Section 5.5 defines algorithms to compute matching prefixes and substrings for an input RE and its associated correctness theorems.

5.1 RE syntax and semantics

Definition of RE syntax as an Agda type is straightforward. We represent alphabet symbols using characters, type Char, and strings as lists of characters.

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

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

RE semantics is defined as an inductive data type in which each constructor represents a rule for the inductive RE semantics.

```
data _∈⟦_⟧ : List Char → Regex → Set where
  λ : [] ∈⟦ λ ⟧
  $_ : (a : Char) → [ a ] ∈⟦ $ a ⟧
  _•_⇒_ : xs ∈⟦ l ⟧ → ys ∈⟦ r ⟧ → zs ≡ xs ++ ys → zs ∈⟦ l • r ⟧
  _+L_ : (r : Regex) → xs ∈⟦ l ⟧ → xs ∈⟦ l + r ⟧
  _+R_ : (l : Regex) → xs ∈⟦ r ⟧ → xs ∈⟦ l + r ⟧
  _⋆ : xs ∈⟦ λ + (e • e ⋆) ⟧ → xs ∈⟦ e ⋆ ⟧
```

Constructor λ states that the empty string (denoted by the empty list [ ]) is in the language of RE λ.

For any single character a, the singleton string $[\ a\ ]$ is in the RL for $\$\ a$. Given string membership proofs for REs l and r, $xs \in [\![\ l\ ]\!]$ and $ys \in [\![\ r\ ]\!]$, constructor $\_ \bullet \_ \Rightarrow \_$ can be used to build a proof for the concatenation of these REs. Note that in the concatenation constructor we introduce an equality $zs \ \equiv\ xs +\!\!+ ys$ that seems unnecessary. This equality simplifies the pattern-matching on proofs of $zs \in [\![\ l \bullet r\ ]\!]$. Constructor $\_ + \mathsf{L}\_$ ($\_ + \mathsf{R}\_$) creates a proof for $l + r$ from a membership proof for l (r). Proofs for Kleene star are built using the following well known equivalence of REs: $e^\star = \lambda + e\,e^\star$.

Several inversion lemmas about RE parsing relation are necessary for derivative-based parsing formalization. They consist of pattern-matching on proofs of $\_ \in [\![\_]\!]$. As an example, below we present the inversion lemma for choice operator.

```
+invert : ∀ {xs l r} → xs ∈ ⟦ l + r ⟧ → xs ∈ ⟦ l ⟧ ∨ xs ∈ ⟦ r ⟧
+invert (r +L pr)  =  inj₁ pr
+invert (l +R pr)  =  inj₂ pr
```

Intuitively, function +invert specifies that if it is the case that $xs \in [\![\ l + r\ ]\!]$ then the string xs matches the RE l or it matches the RE r. Another inversion lemma (function charInvert) specifies that if a string $x :: xs$ matches the RE $\$\ y$ then the input string must be a single character string, i.e. $xs \ \equiv\ [\,]$ and $x \ \equiv\ y$.

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

In our formalization we defined other inversions lemmas for RE semantics relation. They follow the same structure of the previously defined lemmas — they produce, as result, the necessary conditions for a RE semantics proof to hold — and are omitted for brevity.

5.2 Smart-constructors

In order to define Brzozowski derivatives, we follow Owens et al. [30]. We use smart constructors to identify equivalent REs modulo identity and nullable elements, λ and $\emptyset$, respectively. The equivalence axioms maintained by smart constructors are:

- For union:

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

- For concatenation:

$$\begin{array}{ll} 1)\ e\,\emptyset \approx \emptyset & 2)\ e\,\lambda \approx e \\ 3)\ \emptyset\,e \approx \emptyset & 4)\ \lambda\,e \approx e \end{array}$$

- For Kleene star:

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

These axioms are kept as invariants using functions that preserve them while building REs. As a convention, we name smart constructors by prefixing a back quote to the constructor name. For union, we just need to worry when one parameter denotes the empty language:

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

For concatenation, we need to deal with the possibility of parameters being empty (denoting the empty language) or λ. If one of them is empty ($\emptyset$) the result is also empty. Since the empty string is the identity for concatenation, the other parameter is returned.

```
_`•_ : (e e' : Regex)  →  Regex
∅ `• e'  =  ∅
λ `• e'  =  e'
e `• ∅  =  ∅
e `• λ  =  e
e `• e'  =  e • e'
```

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

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

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

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

Proof.

($\rightarrow$) : 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 $\emptyset \text{ `}{+}\ e' = e'$ and the desired result follows. The same reasoning applies for $e' = \emptyset$.

($\leftarrow$) : 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 and completeness of concatenation)**.** *For all REs e, e' and all strings xs, $xs \in [\![\ e \text{ `}{\bullet}\ e'\ ]\!]$ holds if and only if, $xs \in [\![\ e \bullet e'\ ]\!]$ also holds.*

Proof.

(→) : By case analysis on the structure of e, e'. The interesting cases are when e or e' are equal to λ or $\emptyset$. When some of the REs are equal to $\emptyset$, the result follows by contradiction. If one of the REs are equal to λ the desired result is immediate, from the proof term $xs \in [\![\ e \text{ ‘}\bullet\ e'\ ]\!]$, using list concatenation properties.

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

Lemma 4 (Soundness and completeness of Kleene star)**.** *For all REs e and string xs, $xs \in [\![\ e \text{ ‘}\star\ ]\!]$ holds if and only if, $xs \in [\![\ e \star\ ]\!]$ also holds.*

Proof. Both directions follows by straightforward case analysis on e’s structure. □

5.3 Brzozowski’s derivatives formalization

Brzozowski’s definition of derivative uses a nullability test, ν.

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

```
ν[_] : ∀ (e : Regex) → Dec ([ ] ∈ ⟦ e ⟧)
ν[ ∅ ] = no (λ ())
ν[ λ ] = yes λ
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 λ) ⋆)
```

The definition of $\nu[\_]$ uses some of the inversion lemmas about RE semantics. Lemma •invert states that if the empty string is in the language of $l \bullet r$ (where l and r are arbitrary REs) then the empty string belongs to l and r’s languages. Lemma +invert is defined similarly for union. Note that the definition of $nuuu[\_]$ ensures, by construction, its correctness (Lemma 1).

The definition of derivative function has an immediate translation to Agda. Notice that it uses smart constructors to quotient resulting REs with respect to the equivalence axioms presented in Section 5.2 and RE nullability test. In the symbol case (constructor $\$\_$), function $\overset{?}{=}$ is used, which produces an evidence for equality of two Char values.

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

From this definition we prove the following important properties of the derivative operation. Soundness of $\delta[\_,\_]$ ensures that if a string xs is in the language of $\delta[\ e\ ,\ x\ ]$, then $(x :: xs) \in [\![\ e\ ]\!]$ holds. Completeness ensures that the other direction of implication holds.

Theorem 7 (Derivative operation soundness)**.** *For all REs e, all strings xs and all symbols x, if $xs \in [\![\ \delta[\ e\ ,\ x\ ]\ ]\!]$ holds 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 8 (Derivative operation completeness)**.** *For all REs e, all strings xs and all symbols x, if $(x :: xs) \in [\![\ e\ ]\!]$ holds then $xs \in [\![\ \delta[\ e\ ,\ x\ ]\ ]\!]$ holds.*

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

5.4 Antimirov's partial derivatives formalization

A key point in Antimirov's partial derivatives is how to represent sets of REs. Aiming for simplicity, we represent a set of REs as a list:

```
Regexes = List Regex
```

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

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

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 ] = [ ]
∇[ λ , c ] = [ ]
∇[ $ x , c ] with x ≟ c
∇[ $ x , .x ] | yes refl = [ λ ]
∇[ $ x , c ] | no p = [ ]
∇[ e • e′ , c ] with ν[ e ]
∇[ e • e′ , c ] | yes p = (e′ ⊙ ∇[ e , c ]) ++ ∇[ e′ , c ]
∇[ e • e′ , c ] | no ¬p = e′ ⊙ ∇[ e , c ]
∇[ e + e′ , c ] = ∇[ e , c ] ++ ∇[ e′ , c ]
∇[ e ⋆ , c ] = (e ⋆) ⊙ ∇[ e , c ]
```

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

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

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

Lemma 5 (Weakening left)**.** *For all sets of REs E, E' and all strings s, if $s \in \langle\langle\, E \,\rangle\rangle$ holds then $s \in \langle\langle\, E \mathbin{++} E' \,\rangle\rangle$ holds.*

Proof. Straightforward induction on the derivation of $s \in \langle\langle\, E \,\rangle\rangle$. □

Lemma 6 (Weakening right)**.** *For all sets of REs E, E' and all strings s, if $s \in \langle\langle\, E' \,\rangle\rangle$ holds then $s \in \langle\langle\, E \mathbin{++} E' \,\rangle\rangle$ holds.*

Proof. Straightforward induction on the derivation of $s \in \langle\langle\, E' \,\rangle\rangle$. □

Lemma 7. *For all sets of REs E, E' and all strings s, if $s \in \langle\langle\, E \mathbin{++} E' \,\rangle\rangle$ holds then $s \in \langle\langle\, E \,\rangle\rangle$ or $s \in \langle\langle\, E' \,\rangle\rangle$ holds.*

Proof. Induction on the derivation of $s \in \langle\langle\, E \mathbin{++} E' \,\rangle\rangle$ and case analysis on the structure of E and E'. □

Lemma 8. *For all sets of REs E, all REs e and all strings s, s'; if $s \in \langle\langle\, E \,\rangle\rangle$ and $s' \in [\![\, e \,]\!]$ holds then $s \mathbin{++} s' \in \langle\langle\, e \odot E \,\rangle\rangle$ holds.*

Proof. Induction on the derivation of $s \in \langle\langle\, E \,\rangle\rangle$. □

Lemma 9. *For all sets of REs E, all REs e and all strings s, if $s \in \langle\langle\ e \odot E\ \rangle\rangle$ holds then there exist s_1 and s_2 such that $s\ \equiv\ s_1 +\!\!+\ s_2$, $s_1 \in \langle\langle\ E\ \rangle\rangle$ and $s_2 \in [\![\ e\ ]\!]$ holds.*

Proof. Induction on the derivation of $s \in \langle\langle\ e \odot E\ \rangle\rangle$. □

Using these previous results about $\_ \in \langle\langle \_ \rangle\rangle$, 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 9 (Partial derivative operation soundness)**.** *For all symbols a, all strings s and all REs e, if $s \in \langle\langle\ \nabla[\ e\ ,\ a\ ]\ \rangle\rangle$ holds then $(a :: s) \in [\![\ e\ ]\!]$ holds.*

Proof. Induction on the structure of e, using Lemmas 7 and 9. □

Theorem 10 (Partial derivative operation completeness)**.** *For all symbols a, all strings s and all REs e, if $(a :: s) \in [\![\ e\ ]\!]$ holds then $s \in \langle\langle\ \nabla[\ e\ ,\ a\ ]\ \rangle\rangle$ holds.*

Proof. Induction on the structure of e, using Lemmas 5, 6 and 8. □

5.5 Computing prefixes and substrings

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

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

```
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 the following relevant properties of prefixes and substrings.

Lemma 10 (Lemma ¬IsPrefix)**.** *For all REs e, if $[\,] \in [\![\ e\ ]\!]$ does not hold then neither does IsPrefix $[\,]$ e.*

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

Lemma 11 (Lemma ¬IsPrefix− ::)**.** *For all REs* e *and all strings* xs*, if* [ ] ∈ ⟦ e ⟧ *and* IsPrefix xs δ[e , x] *do not hold then neither does* IsPrefix (x :: xs) e*.*

Proof. Immediate from the definition of IsPrefix and Theorem 8. □

Lemma 12 (Lemma ¬IsSubstring)**.** *For all REs* e*, if* IsPrefix [ ] e *does not hold then neither does* IsSubstring [ ] e*.*

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

Lemma 13 (Lemma ¬IsSubstring− ::)**.** *For all strings* xs*, all symbols* x *and all REs* e*, if* IsPrefix (x :: xs) e *and* IsSubstring xs e *do not hold then neither does* IsSubstring (x :: xs) e*.*

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

Function IsPrefixDec decides if a given RE e matches a prefix in xs by induction on the structure of xs, using Lemmas 10, 11, decidable emptyness test ν[_] and Theorem 7. Intuitively, IsPrefixDec first checks if current RE e accepts the empty string. In this case, [ ] is returned as a prefix. Otherwise, it verifies, for each symbol x, whether RE δ[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 11. Note that function ∂ − sound is the soundness theorem for derivatives (theorem 7).

```
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 (δ[ 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 the structure of the input string xs, using IsPrefixDec to check whether it is possible to match a prefix of e. In this case, a substring is built from this prefix. If there's no such prefix, a recursive call is made to check if there is a substring match, returning such substring or a proof that it does not exist.

```
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 relations. Such lemmas are slighly modified versions of Lemmas 10 and 11 that consider the relation _ ∈ ⟨⟨_⟩⟩ and are omitted for brevity.

5.6 Concluding remarks

In this chapter we describe the main points in the formalization of the derivative-based parsing algorithms formalized. We present the main Agda definitions, discuss our design decisions and present proof sketches of the relevant properties.

The complete formalization and instructions on how to build it can be found at the following repository:

`https://www.github.com/raulfpl/regex`

6 Implementation details and experiments

From the formalized algorithm we built a tool for RE parsing in the style of GNU Grep [16]. 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, named verigrep, involved a comparison of its performance with GNU Grep [16] (grep), Google regular expression library re2 [31] and Haskell RE parsing algorithms haskell-regexp, described in [14]. We run RE parsing experiments on a machine with an Intel Core I7-4500U 1.8 GHz, 8GB RAM running Windows 10; the results were collected and the median of several test runs was computed.

We use the same experiments as those used in [35]; 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` and files containing thousands of occurrences of `c` followed by a single `a`, using the same RE. Results are presented in Figures 6.1, 6.2, and 6.3 respectively.

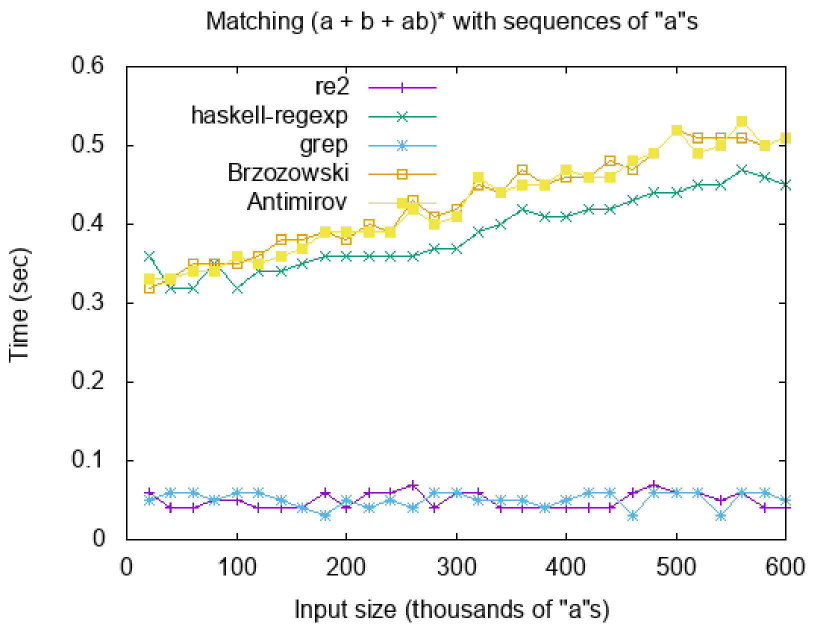


Figure 6.1: Results of experiment 1.

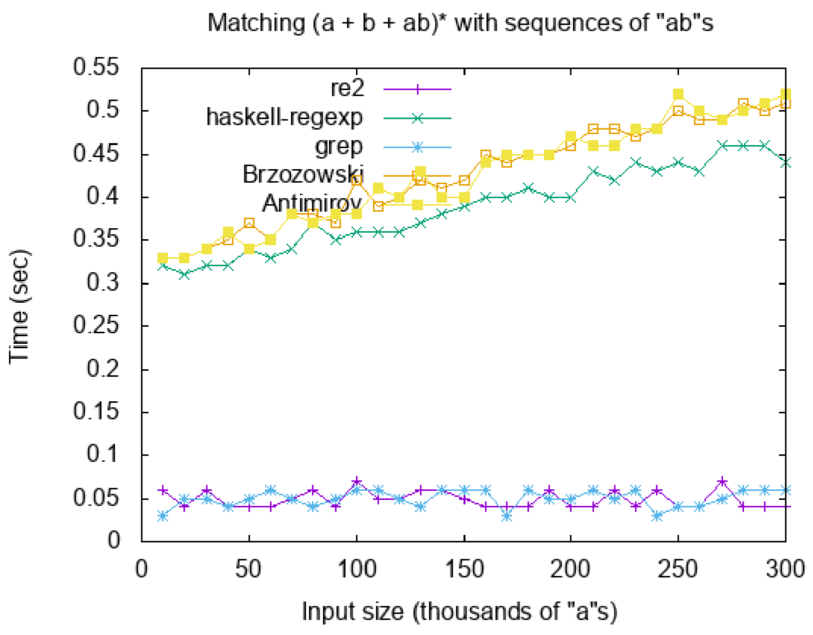


Figure 6.2: Results of experiment 2.

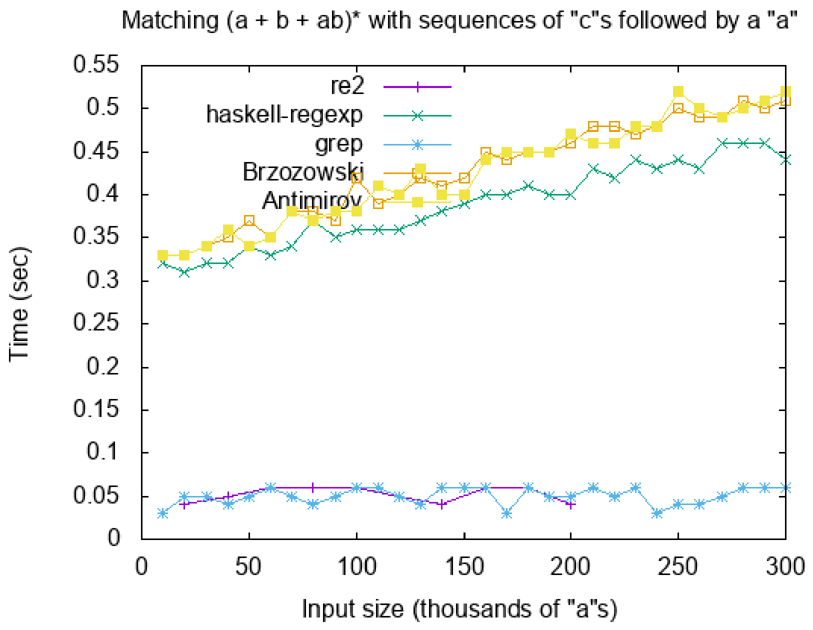


Figure 6.3: Results of experiment 3.

Our tool behaves poorly when compared with all other options considered. The cause of this inefficiency needs further investigation, but we envisaged that it can be due to the following: 1) Our algorithm relies on the Brzozowski's definition for RE parsing, which needs to quotient resulting REs. 2) We use lists to represent sets of Antimirov's partial derivatives. We believe that usage of better data structures to represent sets and using appropriate disambiguation strategies like greedy parsing [15] and POSIX [35] would be able to improve the efficiency of our algorithm without sacrificing correctness. We leave the formalization of disambiguation strategies and the use of more efficient data structures for future work.

All scripts needed to replicate the experiments and instructions on how to compile the verigrep tool are available at the following repository:

`https://www.github.com/raulfpl/regex`

7 Conclusion and future works

We gave 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.

In our view, this work provides an incentive for the adoption of dependently typed programming languages for software development tasks. The possibility of combining programs and proofs allow us to enforce the desired properties of a function during its construction, reducing the necessary costs to debug and test code. In Andrew Appel's words:

> "Purely functional programming, inductive proofs and an expressive type system are the main ingredients for producing correct software."

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 [35, 15], using derivatives and partial derivatives to build finite state automata for RE parsing and investigate other approaches to obtain a formalized but simple and efficient RE parsing tool.

Bibliography

[1] Agda's documentation. `https://agda.readthedocs.io/en/v2.5.2/`.

[2] 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.

[3] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.

[4] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.

[5] Fahad Ausaf, Roy Dyckhoff, and Christian Urban. POSIX lexing with derivatives of regular expressions (proof pearl). In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 69–86. Springer, 2016.

[6] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types.* Cambridge University Press, New York, NY, USA, 2013.

[7] Jean-Philippe Bernardy and Patrik Jansson. Certified context-free parsing: A formalisation of valiant's algorithm in agda. *CoRR*, abs/1601.07724, 2016.

[8] 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.

[9] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.

[10] Nils Anders Danielsson. Total parser combinators. *SIGPLAN Not.*, 45(9):285–296, September 2010.

[11] 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.

[12] 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.

[13] 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).

[14] 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.

[15] 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.

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

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

[18] A. Heyting. *Intuitionism, an Introduction.* North-Holland, 1956.

[19] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, New York, NY, USA, 2 edition, 2008.

[20] 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.

[21] 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.

[22] 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.

[23] Raul Lopes, Rodrigo Ribeiro, and Carlos Camarão. Certified derivative-based parsing of regular expressions. In *Programming Languages — Lecture Notes in Computer Science 9889*, pages 95–109. Springer, 2016.

[24] Hugo Daniel Macedo and José Nuno Oliveira. Typing linear algebra: A biproduct-oriented approach. *CoRR*, abs/1312.4818, 2013.

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

[26] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, January 2004.

[27] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. *SIGPLAN Not.*, 46(9):189–195, September 2011.

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

[29] 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.

[30] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009.

[31] Google Regular Expression Library - re2. `https://github.com/google/re2`.

[32] Tom 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, pages 103–118, Berlin, Heidelberg, 2011. Springer-Verlag.

[33] 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.

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

[35] 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.

[36] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `http://homotopytypetheory.org/book/`, 2013.

[37] Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, April 1975.