

Formalising Automata Theory in Agda
CERTIFIED TRANSLATION FROM REGULAR EXPRESSION TO FINITE AUTOMATA

Wai Tak, Cheung
Student ID: 1465388
Supervisor: Dr. Martín Escardó



Submitted in conformity with the requirements
for the degree of BSc. Computer Science
School of Computer Science
University of Birmingham

Copyright © 2016 School of Computer Science, University of Birmingham

Abstract

Formalising Automata Theory in Agda Certified Translation from Regular Expression to Finite Automata

Wai Tak, Cheung

Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah
blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah
blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah
blah. Blah blah blah. Blah blah blah. Blah blah blah.

Keywords: language, regular expression, finite automata, agda, thompson's construction, powerset
construction, proofs

Acknowledgments

Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah
blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah
blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah
blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah
blah. Blah blah blah. Blah blah blah. Blah blah blah. Blah blah blah.

All software for this project can be found at
<https://codex.cs.bham.ac.uk/svn/projects/2015/wtc488/>

List of Abbreviations

ϵ-NFA	Non-deterministic Finite Automaton with ϵ -transition
NFA	Non-deterministic Finite Automaton
DFA	Deterministic Finite Automaton
MDFA	Minimised Deterministic Finite Automaton

Contents

List of Abbreviations	5
1 Introduction	8
1.1 Motivation	8
1.2 Overview	8
2 Literature Review	9
2.1 Agda	9
2.1.1 Simply Typed Functional Programming	9
2.1.2 Dependent Types	10
2.1.3 Propositions as Types	10
2.1.4 Universe	10
2.1.5 Termination Checker	10
2.2 Related Work	10
2.2.1 Certified Parsing of Regular Languages	10
3 Formalisation in Type Theory	11
3.1 Subsets and Decidable Subsets	11
3.2 Languages	12
3.2.1 Operations on Languages	12
3.3 Regular Languages and Regular Expressions	13
3.4 ϵ -Non Deterministic Finite Automata	14
3.5 Thompson's Construction	16
3.6 Non-deterministic Finite Automata	20
3.7 Removing ϵ -transitions	22
3.8 Deterministic Finite Automata	22
3.9 Powerset Construction	24
3.10 Minimal DFA	24
3.11 Minimising DFA	24
3.12 Myhill-Nerode Theorem	24
4 Evaluation	25
5 Conclusion	26
References	27

1 Introduction

This project aims to study the feasibility of formalising Automata theory in Type theory with the aid of a dependently-typed functional programming language, Agda. Automata theory is a very extensive piece of work; therefore, this project will only focus on the theorems and proofs that are related to the translation between regular expressions and finite automata. This work also gives a brief introduction on how complex and non-trivial proofs are formalised in Type theory. The project can be separated into three parts: 1) translating any regular expressions to a DFA, 2) proving the correctness of the translation and 3) formalising the Myhill-Nerode Theorem. In this stage, we are only interested in the correctness of the translation but not the efficiency.

1.1 Motivation

Parsing has been a very significant area of study throughout the history of computer science. This technique has been used extensively in a variety of disciplines: 1) in computer science, we have compiler construction, artificial intelligence and data mining; 2) in linguistics, we have ... and 3) in chemistry and bioinformatics, we have ...

1.2 Overview

In section two, we will describe the background of proof assistant and to discuss a similar research. We will give a brief introduction on Agda as a programming language and a proof assistant. Then there will be some descriptions of all the types that are frequently used in the project. We will also look into some small Agda proofs so that readers can have a taste of how proofs are formalised in Type theory. In the end of section two, we will look at the research [3] conducted by Denis Firsov and Tarmo Uustalu. Following the literature review, the third section will be a detail description of our work. We will walk through the Agda formalisation of the three objectives. After that, we will evaluate the whole project in terms of ... and Finally, the conclusions will be drawn.

2 Literature Review

2.1 Agda

Agda is a dependently typed functional programming language and a proof assistant based on Intuitionistic Type theory [4]. The current version (Agda 2) is rewritten by Ulf Norell [5] at Chalmers University of Technology. In this section, we will describe the basic features of Agda and how dependent types are employed to build programs and proofs. Most of the materials presented below can also be found in the two tutorial papers [2] and [6]. Interested readers can take a look at them and get a more precise idea on how to work with Agda.

2.1.1 Simply Typed Functional Programming

In this part, we will begin by showing how to do ordinary functional programming in Agda. Haskell is the implementation language of Agda, as we will see below, Agda has borrowed many features from Haskell. In below, we will show how to define basic data types (boolean and natural number) and basic functions over them.

Boolean. We first introduce the type of boolean in Agda:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Bool is a data type having two constructors: *true* and *false*. Note that these two constructors are also elements of *Bool* since they do not take any arguments. Their types are explicitly declared by '*: Bool*'. On the other hand, *Bool* is a member of the type *Set*. *Set* represents the set of all *small types*. *Bool* is a *small type* but *Set* itself is not, it is a *large type*. We will look into the difference in later part. Now, let us define the negation function on boolean values:

```
not : Bool → Bool
not true  = false
not false = true
```

Similar to Haskell or other functional programming languages, we declare the type of the function and define the function body by pattern matching the argument. We can declare partial functions in Haskell but not in Agda. For instance, the function below will be rejected by the Agda compiler:

```
not : Bool → Bool
not true  = false
```

Natural Number. The type of natural numbers is defined inductively as the data type below:

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

Let us define the addition function recursively:

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

Parameterised Types In Haskell, the type of list $[a]$ is parameterised by a type parameter a . The analogous data type in Agda is defined inductively as:

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

2.1.2 Dependent Types

Dependent types are types that depend on values of other types. For example, A^n is the type of vectors with length n . This kind of data type are impossible to be declared in simple-typed system. Now, we define the vector type in Agda inductively as follow:

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

2.1.3 Propositions as Types

Propositional Logic ...

Predicate Logic ...

2.1.4 Universe

...

2.1.5 Termination Checker

...

2.2 Related Work

2.2.1 Certified Parsing of Regular Languages

The matrix representation.

3 Formalisation in Type Theory

Let us recall the three objectives of the project: 1) translating any regular expressions to a DFA 2) proving the correctness of the translation and 3) formalising the Myhill-Nerode Theorem.

In part 1), the translation was divided into the following steps. First, we followed Thompson's construction algorithm to convert any regular expressions to an ϵ -NFA. Then we removed all the ϵ -transitions in the ϵ -NFA by computing the ϵ -closure for every states. After that, we used powerset construction to create a DFA. Finally, we removed all the unreachable states and then used quotient construction to obtain the minimised DFA.

In part 2), the correctness proofs of the above translation were also separated into different steps according to part 1). For each of the translation steps in part 1), we proved that the language accepted by the input is equal to the language accepted by its translated output. i.e. $L(regex) = L(translated \ \epsilon\text{-NFA}) = L(translated \text{ DFA}) = L(translated \text{ MDFA})$.

In part 3), (pending...)

In the following parts, we will walk through the formalisation of each of the above steps together with their correctness proofs. Before we go into the steps, we first need to have a representation of subset as it is a fundamental element in the theory. All of the definitions, theorems, lemmas and proofs written in below correspond to their formalisation in Agda.

3.1 Subsets and Decidable Subsets

Agda Please refers to Subset.agda and Subset/DecidableSubset.agda

Definition 1.1 Suppose A is a set, in Type theory, its subsets are represented as a unary function on A , i.e. $Subset \ A = A \rightarrow Set$.

When declaring a subset in Agda, we can write $SubA = \lambda a \rightarrow ?$, the $?$ here defines the condition for a to be included in $SubA$. This construction is very similar to set comprehension. For example, the subset $\{a \mid a \in A, P(a)\}$ corresponds to $\lambda a \rightarrow P \ a$. Subset is also a unary predicate of A ; therefore, the decidability of it will remain unknown until it is proved.

Definition 1.2 The other representation of subset is $DecSubset \ A = A \rightarrow Bool$. Unlike *Subset*, its decidability is ensured by its definition.

The two definitions have different purposes. *Subset* is used to represent *Language* because not every language is decidable. For other parts such as a subset of states in an automaton, *DecSubset* is used as the decidability is assumed in the definition. The two definitions are defined in Subset.agda

and `Subset/DecidableSubset.agda` respectively as stated at the top. Operators such as membership (\in), subset (\subseteq), superset (\supseteq) and equality ($=$) can also be found in the two files.

Now, by using the representation of subset, we can define languages, regular expressions and finite automata. Their formalisation in this project followed tightly to the definition in [1] written by Aho, A. and Ullman, J.

3.2 Languages

Agda Please refers to `Language.agda`

Suppose we have a set of alphabets Σ ; in Type theory, it can be represented as a data type, i.e. $\Sigma : \text{Set}$. Notice that the decidable equality of Σ is assumed. In Agda, they are passed to every modules as parameters $(\Sigma : \text{Set}) (dec : \text{DecEq } \Sigma)$.

Definition 2.1 We first define Σ^* as the set of all strings over Σ . In our approach, it was expressed as a list of Σ , i.e. $\Sigma^* = \text{List } \Sigma$.

For example, $(A :: g :: d :: a :: [])$ represents the string 'Agda' and the empty list $[]$ represents the empty string ϵ . In this way, we can pattern match on the input string in order to get the first input alphabet and to run a transition from a particular state to another state.

Definition 2.2 A language is a subset of Σ^* ; in Type theory, $\text{Language} = \text{Subset } \Sigma^*$. Notice that *Subset* instead of *DecSubset* is used because not every language is decidable.

3.2.1 Operations on Languages

Definition 2.3 If L_1 and L_2 are languages, then the union of the two languages $L_1 \cup L_2$ is defined as $\{w \mid w \in L_1 \vee w \in L_2\}$. In Type theory, we define it as $L_1 \cup L_2 = \lambda w \rightarrow w \in L_1 \uplus w \in L_2$.

Definition 2.4 If L_1 and L_2 are languages, then the concatenation of the two languages $L_1 \bullet L_2$ is defined as $\{w \mid \exists u \in L_1. \exists v \in L_2. w = uv\}$. In Type theory, we define it as $L_1 \bullet L_2 = \lambda w \rightarrow \exists [u \in \Sigma^*] \exists [v \in \Sigma^*] (u \in L_1 \times v \in L_2 \times w \equiv u ++ v)$.

Definition 2.5 If L is a language, then the closure of L , L^* is defined as $\bigcup_{n \in \mathbb{N}} L^n$ where $L^n = L \bullet L^{n-1}$ and $L^0 = \{\epsilon\}$. In Type theory, we have $L^* = \lambda w \rightarrow \exists [n \in \mathbb{N}] (w \in L^{\wedge} n)$ where the function $_{\wedge}$ is defined recursively as:

$$\begin{aligned} _^{\wedge} _ &: \text{Language} \rightarrow \text{Language} \rightarrow \text{Language} \\ L^{\wedge} \text{ zero} &= [\epsilon] \\ L^{\wedge} (\text{suc } n) &= L \bullet L^{\wedge} n \end{aligned}$$

3.3 Regular Languages and Regular Expressions

Agda Please refers to RegularExpression.agda

Definition 3.1 We define regular languages over Σ inductively as follow:

1. \emptyset is a regular language;
2. $\{\epsilon\}$ is a regular language;
3. $\forall a \in \Sigma. \{a\}$ is a regular language;
4. if L_1 and L_2 are regular languages, then
 - (a) $L_1 \cup L_2$ is a regular language;
 - (b) $L_1 \bullet L_2$ is a regular language;
 - (c) $L_1 \star$ is a regular language.

Listing 1: Regular languages

```
data Regular : Language → Set1 where
  nullL : ∀ {L} → L ≈ ∅ → Regular L
  empty : ∀ {L} → L ≈ [ε] → Regular L
  singl : ∀ {L} → (a : Σ) → L ≈ [a] → Regular L
  union : ∀ {L} L1 L2 → Regular L1 → Regular L2 → L ≈ L1 ∪ L2 → Regular L
  conca : ∀ {L} L1 L2 → Regular L1 → Regular L2 → L ≈ L1 • L2 → Regular L
  kleen : ∀ {L} L1 → Regular L1 → L ≈ L1 ★ → Regular L
```

Definition 3.2 Here we define regular expressions inductively over Σ as follow:

1. \emptyset is a regular expression denoting the regular language \emptyset ;
2. ϵ is a regular expression denoting the regular language $\{\epsilon\}$;
3. $\forall a \in \Sigma. a$ is a regular expression denoting the regular language $\{a\}$;
4. if e_1 and e_2 are regular expressions denoting the regular languages L_1 and L_2 respectively, then
 - (a) $e_1 \mid e_2$ is a regular expressions denoting the regular language $L_1 \cup L_2$;
 - (b) $e_1 \cdot e_2$ is a regular expression denoting the regular language $L_1 \bullet L_2$;
 - (c) e_1^* is a regular expression denoting the regular language $L_1 \star$.

The Agda formalisation is separated into two parts, firstly the definition of regular expressions and secondly the languages denoted by them.

Listing 2: Regular expressions

```
data RegExp : Set where
```

$$\begin{aligned}
\emptyset & : \text{RegExp} \\
\epsilon & : \text{RegExp} \\
\sigma & : \Sigma \rightarrow \text{RegExp} \\
_ | _ & : \text{RegExp} \rightarrow \text{RegExp} \rightarrow \text{RegExp} \\
_ \cdot _ & : \text{RegExp} \rightarrow \text{RegExp} \rightarrow \text{RegExp} \\
_ ^* & : \text{RegExp} \rightarrow \text{RegExp}
\end{aligned}$$

Listing 3: Languages denoted by regular expressions

$$\begin{aligned}
L^R & : \text{RegExp} \rightarrow \text{Language} \\
L^R \emptyset & = \emptyset \\
L^R \epsilon & = \llbracket \epsilon \rrbracket \\
L^R (\sigma \ a) & = \llbracket a \rrbracket \\
L^R (e_1 \mid e_2) & = L^R e_1 \cup L^R e_2 \\
L^R (e_1 \cdot e_2) & = L^R e_1 \bullet L^R e_2 \\
L^R (e^*) & = (L^R e) \star
\end{aligned}$$

3.4 ϵ -Non Deterministic Finite Automata

Agda Please refers to `eNFA.agda`

By now, the set of strings we have considered are in the form of $List \ \Sigma^*$. However, this definition gives us no way to extract an ϵ -transition from the input string. Therefore, we need to introduce another representation of the set of strings specifically for this purpose. (For Definition 4.1 and 4.2, please refers to `Language.agda`)

Definition 4.1 We define Σ^e as the union of Σ and $\{\epsilon\}$, i.e. $\Sigma^e = \Sigma \cup \{\epsilon\}$. In Agda, this can be expressed by a data type definition:

$$\begin{aligned}
& \text{data } \Sigma^e : \text{Set} \text{ where} \\
& \quad \alpha : \Sigma \rightarrow \Sigma^e \\
& \quad E : \Sigma^e
\end{aligned}$$

Definition 4.2 Now we define Σ^{e*} , the set of all strings over Σ^e in a way similar to Σ^* , i.e. $\Sigma^{e*} = List \ \Sigma^e$.

For example, the string 'Agda' can be represented by $(\alpha \ A :: \alpha \ g :: E :: \alpha \ d :: E :: \alpha \ a :: [])$ or $(E :: \alpha \ A :: E :: E :: \alpha \ g :: \alpha \ d :: E :: \alpha \ a :: [])$. We say that these two lists are ϵ -strings of the word 'Agda'. When pattern matching on an ϵ -string, we can know if there is an ϵ -transition or

not. Other operators and lemmas regarding ϵ -strings such as $to\Sigma^* : \Sigma^{e*} \rightarrow \Sigma^*$ can also be found in `Language.agda`.

Now, let us define ϵ -NFA.

Definition 4.3 An ϵ -NFA is a 5-tuple $M = (Q, \Sigma^e, \delta, q_0, F)$, where

1. Q is a finite set of states;
2. Σ^e is the union of Σ and $\{\epsilon\}$;
3. δ is a mapping from $Q \times \Sigma^e$ to $\mathcal{P}(Q)$ which defines the behaviour of the automata;
4. q_0 in Q is the initial state;
5. $F \subseteq Q$ is the set of accepting states.

Listing 4: ϵ -NFA

```
record  $\epsilon$ -NFA : Set1 where
  field
    Q      : Set
     $\delta$     : Q  $\rightarrow$   $\Sigma^e \rightarrow$  DecSubset Q
    q0    : Q
    F      : DecSubset Q
     $\forall qEq$  :  $\forall q \rightarrow q \in^d \delta q E$ 
    Q?     : DecEq Q
    |Q|-1  :  $\mathbb{N}$ 
    It     : Vec Q (suc |Q|-1)
     $\forall q \in It$  : (q : Q)  $\rightarrow$  (q  $\in^V$  It)
    unique : Unique It
```

The set of alphabets $\Sigma : Set$ is passed to the file parameters. Together with Q , δ , q_0 and F , these five fields correspond to the 5-tuple ϵ -NFA. $\forall qEq$ is a proof that any state in Q can reach itself by an ϵ -transition. $Q?$ is the decidable equality of Q . $|Q| - 1$ is the number of states - 1. ' It ' is a vector of length $|Q|$ containing all the states in Q . $\forall q \in It$ is a proof that all states in Q are also in the vector ' It '. *unique* is a proof that there is no repeating elements in ' It '. These extra fields are important when computing ϵ -closures, we will look into them again later in more details.

Now, we want to define the set of strings Σ^* accepted by a given ϵ -NFA. However, before we can do this, we have to define some operations.

Definition 4.4 A configuration is a pair $Q \times \Sigma^{e*}$. Notice that the configuration is based on Σ^{e*} but not Σ^* .

Definition 4.5 A move by an ϵ -NFA N is represented by a binary function \vdash on configurations. We say that $(q, aw) \vdash (q', w)$ for all w in Σ^{e*} if and only if $q' \in \delta(q, a)$ where $a \in \Sigma^e$. In Agda, we have

$$\begin{aligned} _ \vdash _ & : (\mathbb{Q} \times \Sigma^e \times \Sigma^{e*}) \rightarrow (\mathbb{Q} \times \Sigma^{e*}) \rightarrow \text{Set} \\ (q \ , \ a \ , \ w) \vdash (q' \ , \ w') & = w \equiv w' \times q' \in^d \delta \ q \ a \end{aligned}$$

Definition 4.6 We say that $C \vdash^0 C'$ if and only if $C = C'$. We say that $C_0 \vdash^k C_k$ for any $k \geq 1$ if and only if there exists a chain of configurations C_1, C_2, \dots, C_{k-1} such that $C_i \vdash C_{i+1}$ for all $0 \leq i \leq k$. In Agda, we define it recursively as

$$\begin{aligned} _ \vdash^k _ & : (\mathbb{Q} \times \Sigma^{e*}) \rightarrow \mathbb{N} \rightarrow (\mathbb{Q} \times \Sigma^{e*}) \rightarrow \text{Set} \\ (q \ , \ w^e) \vdash^k \text{zero} & - (q' \ , \ w'^e) \\ & = q \equiv q' \times w^e \equiv w'^e \\ (q \ , \ w^e) \vdash^k \text{suc } n & - (q' \ , \ w'^e) \\ & = \exists [p \in \mathbb{Q}] \ \exists [a^e \in \Sigma^e] \ \exists [u^e \in \Sigma^{e*}] \\ & \quad (w^e \equiv a^e :: u^e \times (q \ , \ a^e \ , \ u^e) \vdash (p \ , \ u^e) \times (p \ , \ u^e) \vdash^k n - (q' \ , \ w'^e)) \end{aligned}$$

Definition 4.7 We say that $C \vdash^* C'$ if and only if there exists a number of chains n such that $C \vdash^n C'$. In Agda, we have

$$\begin{aligned} _ \vdash^* _ & : (\mathbb{Q} \times \Sigma^{e*}) \rightarrow (\mathbb{Q} \times \Sigma^{e*}) \rightarrow \text{Set} \\ (q \ , \ w^e) \vdash^* (q' \ , \ w'^e) & = \exists [n \in \mathbb{N}] \ (q \ , \ w^e) \vdash^k n - (q' \ , \ w'^e) \end{aligned}$$

Definition 4.8 For any string w , it is accepted by an ϵ -NFA N if and only if there exists a chain of configurations from q_0, w^e to q, ϵ where w^e is an ϵ -string of w and $q \in F$.

Definition 4.9 The language accepted by an ϵ -NFA is given by the set $\{ w \mid \exists w^e \in \Sigma^{e*}. w = \text{to}\Sigma^*(w^e) \wedge \exists q \in F. (q_0 \ , \ w^e) \vdash^* (q \ , \ \epsilon) \}$. In Agda, we have

$$\begin{aligned} L^{eN} & : \epsilon\text{-NFA} \rightarrow \text{Language} \\ L^{eN} \text{ nfa} & = \lambda w \rightarrow \\ & \quad \exists [w^e \in \Sigma^{e*}] (w \equiv \text{to}\Sigma^* w^e \times (\exists [q \in \mathbb{Q}] (q \in^d F \times (q_0 \ , \ w^e) \vdash^* (q \ , \ [])))) \end{aligned}$$

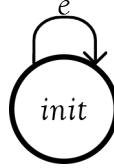
Now that we have the definition of regular expressions and ϵ -NFA, we can formulate the translation using Thompson's Construction.

3.5 Thompson's Construction

Agda Please refers to State.agda and the function **regexTo ϵ -NFA** in Translation.agda

Definition 5.1 The translation for any regular expressions to an ϵ -NFA is defined inductively as follow:

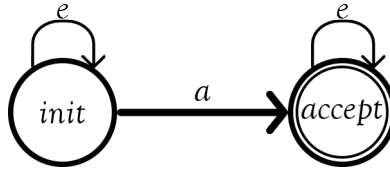
1. for \emptyset , we have $M = (\{init\}, \Sigma^e, \delta, init, \emptyset)$ and graphically



2. for ϵ , we have $M = (\{init\}, \Sigma^e, \delta, init, \{init\})$ and graphically

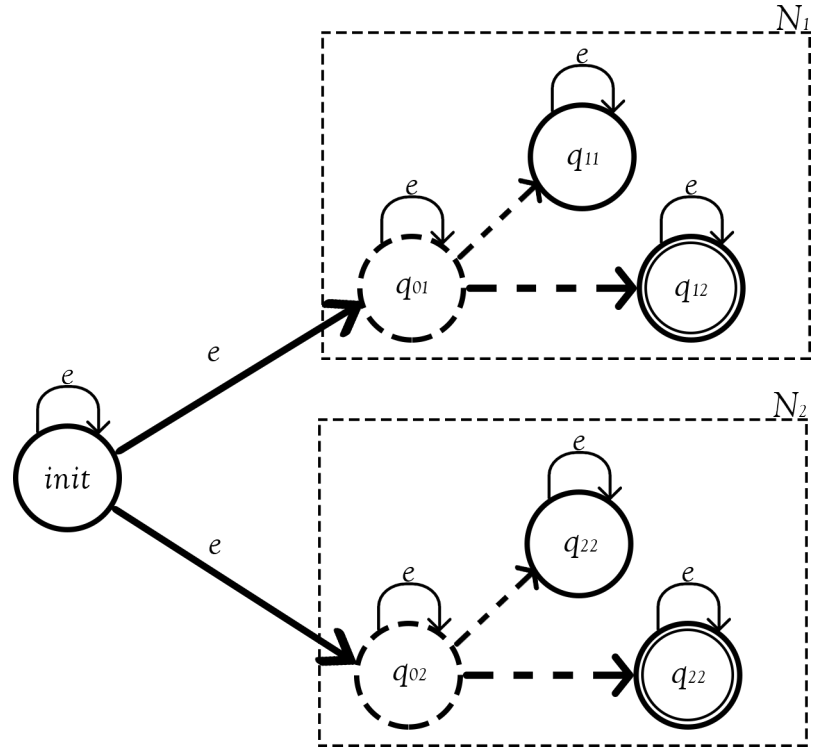


3. for a , we have $M = (\{init, accept\}, \Sigma^e, \delta, init, \{accept\})$ and graphically

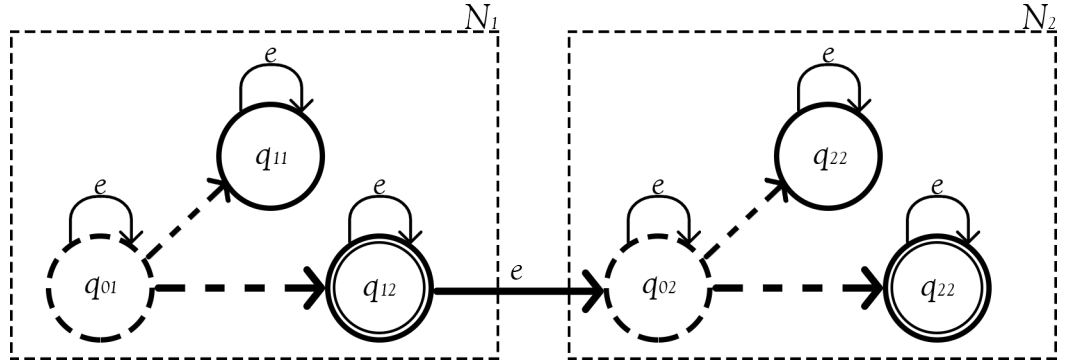


4. if $N_1 = (Q_1, \delta_1, q_{01}, F_1)$ and $N_2 = (Q_2, \delta_2, q_{02}, F_2)$ are ϵ -NFAs translated from the regular expressions e_1 and e_2 respectively, then

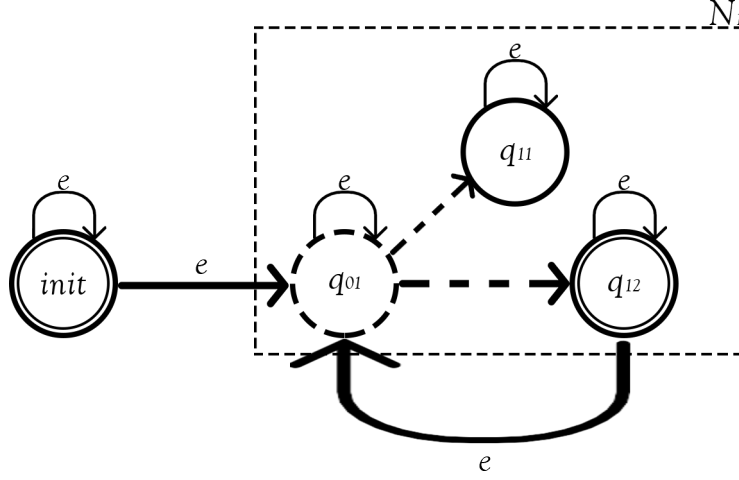
- (a) for $(e_1 \mid e_2)$, we have $M = (\{init\} \cup Q_1 \cup Q_2, \Sigma^e, \delta, init, F_1 \cup F_2)$ and graphically



(b) for $e_1 \cdot e_2$, we have $M = (Q_1 \cup \{mid\} \cup Q_2, \Sigma^e, \delta, init, F_2)$ and graphically



(c) for e_1^* , we have $M = (\{init\} \cup Q_1, \Sigma^e, \delta, init, \{init\} \cup F_1)$ and graphically



Theorem 1.1 For any given regular expressions, its accepted language is equal to the language accepted by its translated ϵ -NFA using Thompson's Construction. i.e. $L(e) = L(\text{translated } \epsilon\text{-NFA})$.

Agda Please refers to the function $\mathbf{L}^R \approx \mathbf{L}^{eN}$ in Correctness.agda

Proof 1.1 We have to prove that for any regular expressions e , $L(e) \subseteq L(\text{translated } \epsilon\text{-NFA})$ and $L(e) \supseteq L(\text{translated } \epsilon\text{-NFA})$ by induction on e .

Base cases: For \emptyset , ϵ and a , by Definition 5.1, it is obvious that the language accepted by them are equal to the language accepted by their translated ϵ -NFA.

Induction hypothesis: For any regular expressions e_1 and e_2 , let $N_1 = (Q_1, \delta_1, q_{01}, F_1)$ and $N_2 = (Q_2, \delta_2, q_{02}, F_2)$ be their translated ϵ -NFA using Definition 5.1 respectively. Then we assume that $L(e_1) = L(N_1)$ and $L(e_2) = L(N_2)$.

Inductive steps:

1) For $(e_1 \mid e_2)$, let $M = (Q, \delta, q_0, F) = (\{init\} \cup Q_1 \cup Q_2, \delta, init, F_1 \cup F_2)$ be its translated ϵ -NFA using Definition 5.1. Then for any string w ,

1.1) if $(e_1 \mid e_2)$ accepts w , by Definition 3.2, either i) e_1 accepts w or ii) e_2 accepts w . Assuming case i), then by induction hypothesis, N_1 also accepts w which also implies that there exists a chain $(q_{01}, w^e) \vdash^* (q, \epsilon)$ in N_1 such that w^e is an ϵ -string of w and $q \in F_1$. Now, we can add an ϵ -transition from $init$ to q_{01} in M such that $(init, \epsilon w^e) \vdash^* (q, \epsilon)$ because $q_{01} \in \delta init \epsilon$. Now, since $q \in F_1$ implies that $q \in F$ and ϵw^e is also an ϵ -string of w ; therefore $w \in L(M)$. The same argument also applies for the case when e_2 accepts w . Since we have proved that $w \in L(e_1 \mid e_2) \Rightarrow w \in L(M)$; therefore $L(e_1 \mid e_2) \subseteq L(M)$ also follows;

1.2) if M accepts w , then there must exists a chain $(init, w^e) \vdash^* (q, \epsilon)$ in M such that w^e is an ϵ -string of w and $q \in F$. Since $q \in F$, therefore $q \neq init$. By Definition 5.1, there are only two possible

ways for $init$ to reach q , via q_{01} or ii) q_{02} . Assuming case i), then we have $(init, \epsilon^+ w_1) \vdash^* (q_{01}, w_1)$ and $(q_{01}, w_1) \vdash^* (q, \epsilon)$ where $w^e = \epsilon^+ w_1$ and $q \in Q_1$. Since we have $q \in F$ and $q \in Q_1$; therefore we have $q \in F_1$. Also w_1 is also an ϵ -string of w , thus the chain $(q_{01}, w_1) \vdash^* (q, \epsilon)$ implies that $w \in L(N_1)$. By induction hypothesis, we have $w \in L(e_1)$ and thus $w \in L(e_1 \mid e_2)$. The same argument also applies for case ii). Since we have proved that $w \in L(M) \Rightarrow w \in L(e_1 \mid e_2)$; therefore $L(e_1 \mid e_2) \supseteq L(M)$ also follows;

1.3) combining 1.1 and 1.2, we have $L(e_1 \mid e_2) = L(M)$.

2) For $(e_1 \cdot e_2)$, let $M = (Q, \delta, q_0, F) = (Q_1 \cup \{mid\} \cup Q_2, \delta, q_{01}, F_2)$ be its translated ϵ -NFA using Definition 5.1. Then for any string w ,

2.1) if $(e_1 \cdot e_2)$ accepts w , then by Definition 3.2, there exists a $u \in L(e_1)$ and a $v \in L(e_2)$ such that $w = uv$. By induction hypothesis, $u \in L(e_1)$ implies that $u \in L(N_1)$ and $v \in L(e_2)$ implies that $v \in L(N_2)$. So there exists a chain: i) $(q_{01}, u^e) \vdash^* (q_1, \epsilon)$ in N_1 where u^e is an ϵ -string of u and $q_1 \in F_1$ and ii) $(q_{02}, v^e) \vdash^* (q_2, \epsilon)$ in N_2 where v^e is an ϵ -string of v and $q_2 \in F_2$. Now we can add an ϵ -transition from q_1 to mid and from mid to q_{02} in order to construct a chain in M . Since $q_2 \in F_2$ implies that $q_2 \in F$ and $u^e v^e$ is an ϵ -string of w implies that so is $u^e \epsilon v^e$; therefore $w \in L(M)$. Since we have proved that $w \in L(e_1 \cdot e_2) \Rightarrow w \in L(M)$, therefore $L(e_1 \cdot e_2) \subseteq L(M)$ also follows;

2.2) if M accepts w , then by Definition 5.1, there must exists a chain $(init, w^e) \vdash^* (q, \epsilon)$ in M where w^e is an ϵ -string of w and $q \in F$. Since $q \in F$, so q must also be in Q_2 . The only possible way for q_{01} to reach q is to go through mid . This implies that there exists a $q_1 \in Q_1$, a $u^e \in \Sigma^{e*}$ and a $v^e \in \Sigma^{e*}$ such that $(q_{01}, u^e \epsilon^+ \epsilon^+ v^e) \vdash^* (q_1, \epsilon^+ \epsilon^+ v^e)$, $q_1 \in F_1$, $(q_{02}, v^e) \vdash^* (q_2, \epsilon)$ and $w^e = u^e \epsilon^+ \epsilon^+ v^e$. Let u and v be the strings represented by u^e and v^e respectively, we have $u \in L(N_1)$ and $v \in L(N_2)$. Then, by induction hypothesis, $u \in L(e_1)$ and $v \in L(e_2)$. Since w^e is an ϵ -string of w , so is $u^e v^e$ and thus $w = uv$. From this, we can deduce that $w \in L(e_1 \cdot e_2)$. Since we have proved that $w \in L(M) \Rightarrow w \in L(e_1 \cdot e_2)$, therefore $L(e_1 \cdot e_2) \supseteq L(M)$ also follows;

2.3) combining 2.1 and 2.2, we have $L(e_1 \cdot e_2) = L(M)$.

3) For e^* , let $M = (Q, \delta, q_0, F) = (Q_1 \cup \{mid\} \cup Q_2, \delta, q_{01}, F_2)$ be its translated ϵ -NFA using Definition 5.1. Then for any string w ,

3.1) if (e^*) accepts w , then there must exists a number n such that $w \in (L \hat{ } n)$. Now, lets do induction on n . **Base case:** when $n = 0$, $L \hat{ } 0 = \dots$

3.2) if M accepts w , ...

3.3) combining 3.1 and 3.2, we have $L(e_1^*) = L(M)$. \square

3.6 Non-deterministic Finite Automata

Agda Please refers to NFA.agda

Definition 6.1 A NFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states;
2. Σ is the set of alphabets;
3. δ is a mapping from $Q \times \Sigma$ to $\mathcal{P}(Q)$ which defines the behaviour of the automata;
4. q_0 in Q is the initial state;
5. $F \subseteq Q$ is the set of accepting states.

Listing 5: NFA

```

record NFA : Set1 where
  field
    Q      : Set
    δ      : Q → Σ → DecSubset Q
    q0    : Q
    F      : DecSubset Q
    Q?     : DecEq Q
    |Q|-1  : ℕ
    It     : Vec Q (suc |Q|-1)
    ∀q∈It  : (q : Q) → (q ∈V It)
    unique : Unique It

```

The set of alphabets $\Sigma : Set$ is passed to the file parameters. Together with Q , δ , q_0 and F , these five fields correspond to the 5-tuple ϵ -NFA. $Q?$ is the decidable equality of Q . $|Q| - 1$ is the number of states - 1. ' It ' is a vector of length $|Q|$ containing all the states in Q . $\forall q \in It$ is a proof that all states in Q are also in the vector ' It '. *unique* is a proof that there is no repeating elements in ' It '. These extra fields are important when computing ϵ -closures, we will look into them again later in more details.

Now, we want to define the set of strings Σ^* accepted by a given NFA. However, before we can do this, we have to define some operations.

Definition 6.2 A configuration is a pair $Q \times \Sigma^*$.

Definition 6.3 A move by an ϵ -NFA N is represented by a binary function \vdash on configurations. We say that $(q, aw) \vdash (q', w)$ for all w in Σ^* if and only if $q' \in \delta(q, a)$ where $a \in \Sigma$. In Agda, we have

$$\begin{aligned}
 & _ \vdash _ : (Q \times \Sigma \times \Sigma^*) \rightarrow (Q \times \Sigma^*) \rightarrow Set \\
 & (q, a, w) \vdash (q', w') = w \equiv w' \times q' \in^d \delta \ q \ a
 \end{aligned}$$

Definition 6.4 We say that $C \vdash^0 C'$ if and only if $C = C'$. We say that $C_0 \vdash^k C_k$ for any $k \geq 1$ if and only if there exists a chain of configurations C_1, C_2, \dots, C_{k-1} such that $C_i \vdash C_{i+1}$ for

all $0 \leq i \leq k$. In Agda, we define it recursively as

$$\begin{aligned}
& _ \vdash^k _ - _ : (Q \times \Sigma^*) \rightarrow \mathbb{N} \rightarrow (Q \times \Sigma^*) \rightarrow \text{Set} \\
& (q, w) \vdash^k \text{zero} - (q', w') \\
& \quad = q \equiv q' \times w \equiv w' \\
& (q, w) \vdash^k \text{suc } n - (q', w') \\
& \quad = \exists [p \in Q] \exists [a \in \Sigma] \exists [u \in \Sigma^*] \\
& \quad \quad (w \equiv a :: u \times (q, a, u) \vdash (p, u) \times (p, u) \vdash^k n - (q', w'))
\end{aligned}$$

Definition 6.5 We say that $C \vdash^* C'$ if and only if there exists a number of chains n such that $C \vdash^n C'$. In Agda, we have

$$\begin{aligned}
& _ \vdash^* _ : (Q \times \Sigma^*) \rightarrow (Q \times \Sigma^*) \rightarrow \text{Set} \\
& (q, w) \vdash^* (q', w') = \exists [n \in \mathbb{N}] (q, w) \vdash^k n - (q', w')
\end{aligned}$$

Definition 6.6 For any string w , it is accepted by an NFA N if and only if there exists a chain of configurations from q_0, w to q, ϵ where $q \in F$.

Definition 6.7 The language accepted by an NFA is given by the set $\{ w \mid \exists q \in F. (q_0, w) \vdash^* (q, \epsilon) \}$. In Agda, we have

$$\begin{aligned}
& L^N : \text{NFA} \rightarrow \text{Language} \\
& L^N \text{ nfa} = \lambda w \rightarrow \exists [q \in Q] (q \in^d F \times (q_0, w) \vdash^* (q, []))
\end{aligned}$$

3.7 Removing ϵ -transitions

...

3.8 Deterministic Finite Automata

Agda Please refers to DFA.agda

Definition 8.1 A DFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states;
2. Σ is the set of alphabets;
3. δ is a mapping from $Q \times \Sigma$ to Q which defines the behaviour of the automata;
4. q_0 in Q is the initial state;
5. $F \subseteq Q$ is the set of accepting states.

Listing 6: NFA

```

record DFA : Set1 where
  field
    Q          : Set
    δ          : Q → Σ → Q
    q0       : Q
    F          : DecSubset Q
    _≈_        : Q → Q → Set
    Dec≈       : ∀ q q' → Dec (q ≈ q')
    ≈isEquiv  : IsEquivalence _≈_
    δ-lem      : ∀ {q} {p} a → q ≈ p → δ q a ≈ δ p a
    F-lem      : ∀ {q} {p} → q ≈ p → q ∈d F → p ∈d F
    Q?         : DecEq Q
    |Q|-1      : ℕ
    It         : Vec Q (suc |Q|-1)
    ∀q∈It      : (q : Q) → (q ∈V It)
    unique     : Unique It

```

The set of alphabets $\Sigma : \text{Set}$ is passed to the file parameters. Together with Q , δ , q_0 and F , these five fields correspond to the 5-tuple ϵ -NFA. $Q?$ is the decidable equality of Q . $|Q| - 1$ is the number of states - 1. ' It ' is a vector of length $|Q|$ containing all the states in Q . $\forall q \in It$ is a proof that all states in Q are also in the vector ' It '. *unique* is a proof that there is no repeating elements in ' It '. These extra fields are important when computing ϵ -closures, we will look into them again later in more details.

Now, we want to define the set of strings Σ^* accepted by a given NFA. However, before we can do this, we have to define some operations.

Definition 6.2 A configuration is a pair $Q \times \Sigma^*$.

Definition 6.3 A move by an ϵ -NFA N is represented by a binary function \vdash on configurations. We say that $(q, aw) \vdash (q', w)$ for all w in Σ^* if and only if $q' \in \delta(q, a)$ where $a \in \Sigma$. In Agda, we have

$$\begin{aligned}
 & \vdash_- : (Q \times \Sigma \times \Sigma^*) \rightarrow (Q \times \Sigma^*) \rightarrow \text{Set} \\
 & (q, a, w) \vdash (q', w') = w \equiv w' \times q' \in^d \delta \ q \ a
 \end{aligned}$$

Definition 6.4 We say that $C \vdash^0 C'$ if and only if $C = C'$. We say that $C_0 \vdash^k C_k$ for any $k \geq 1$ if and only if there exists a chain of configurations C_1, C_2, \dots, C_{k-1} such that $C_i \vdash C_{i+1}$ for

all $0 \leq i \leq k$. In Agda, we define it recursively as

$$\begin{aligned}
& _ \vdash^k _ - _ : (Q \times \Sigma^*) \rightarrow \mathbb{N} \rightarrow (Q \times \Sigma^*) \rightarrow \text{Set} \\
& (q \ , \ w) \vdash^k \text{zero} - (q' \ , \ w') \\
& \quad = q \equiv q' \times w \equiv w' \\
& (q \ , \ w) \vdash^k \text{suc } n - (q' \ , \ w') \\
& \quad = \exists[\ p \in Q \] \ \exists[\ a \in \Sigma \] \ \exists[\ u \in \Sigma^* \] \\
& \quad \quad (w \equiv a :: u \times (q \ , \ a \ , \ u) \vdash (p \ , \ u) \times (p \ , \ u) \vdash^k n - (q' \ , \ w'))
\end{aligned}$$

Definition 6.5 We say that $C \vdash^* C'$ if and only if there exists a number of chains n such that $C \vdash^n C'$. In Agda, we have

$$\begin{aligned}
& _ \vdash^* _ : (Q \times \Sigma^*) \rightarrow (Q \times \Sigma^*) \rightarrow \text{Set} \\
& (q \ , \ w) \vdash^* (q' \ , \ w') = \exists[\ n \in \mathbb{N} \] \ (q \ , \ w) \vdash^k n - (q' \ , \ w')
\end{aligned}$$

Definition 6.6 For any string w , it is accepted by an NFA N if and only if there exists a chain of configurations from q_0, w to q, ϵ where $q \in F$.

Definition 6.7 The language accepted by an NFA is given by the set $\{ w \mid \exists q \in F. (q_0 \ , \ w) \vdash^* (q \ , \ \epsilon) \}$. In Agda, we have

$$\begin{aligned}
& L^N : \text{NFA} \rightarrow \text{Language} \\
& L^N \text{ nfa} = \lambda w \rightarrow \exists[\ q \in Q \] \ (q \in^d F \times (q_0 \ , \ w) \vdash^* (q \ , \ []))
\end{aligned}$$

3.9 Powerset Construction

...

3.10 Minimal DFA

...

3.11 Minimising DFA

...

3.12 Myhill-Nerode Theorem

...

4 Evaluation

How easy/difficult to write proofs in Agda, compare to proofs written in paper. In terms of management, in terms of technical issue?

5 Conclusion

...

References

- [1] Aho, A. V. and Ullman, J. D. (1972). *The Theory of Parsing, Translation and Compiling. Volume I: Parsing*. United States of America: Prentice-Hall International, Inc.
- [2] Bove, A. and Dybjer, P. (2009). Dependent Types at Work. In: Bove, A., Barbosa, L., Pardo, A. and Pinto, J. *Language Engineering and Rigorous Software Development*. Berlin: Springer Berlin Heidelberg. p57-99.
- [3] Firsov, D. and Uustalu, T. (2013). Certified Parsing of Regular Languages. In: Gonthier, G. and Norrish, M. *Certified Programs and Proofs*. Melbourne: Springer International Publishing. p98-113.
- [4] Martin-Lof, P. (1984), *Intuitionistic Type Theory*.
- [5] Norell, U. (2007). *Towards a practical programming language based on dependent type theory*, Goteborg.
- [6] Norell, U. and Chapman, J. (2009). Dependently Typed Programming in Agda. In: Koopman, P., Plasmeijer, R. and Swierstra, D. *Advanced Functional Programming*. Berlin: Springer Berlin Heidelberg. p230-266.
- [7] Norell, U. et al (2016), Available at: <http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage> (Accessed: 12th March 2016).

Appendices

Agda Code?