Validated Parsing of Regular Expressions in Agda

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

Abstract

Validated Parsing of Regular Expressions in Agda

Wai Tak, Cheung

Blah blah blah. Blah blah blah.

Keywords: regular expression, finite automata, agda, proof assistant

Acknowledgments

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

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

Contents

1 Introduction

This project aims to study the feasibility of formalising Automata Theory [2] in Type Theory [12] with the aid of a dependently-typed functional programming language, Agda [17]. Automata Theory is an extensive work; therefore, it will be unrealistic to include all the materials under the time constraints. Accordingly, this project will only focus on the theorems and proofs that are related to the translation of regular expressions to finite automata. In addition, this project also serves as an example of how complex and non-trivial proofs are formalised.

Our Agda formalisation is consist of two components: 1) the translation of regular expressions to DFA and 2) the correctness proofs of the translation. At this stage, we are only interested in the correctness of the translation but not the efficiency of the algorithms.

1.1 Motivation

My motivation on this project is to learn and apply dependent types in formalising programming logic. At the beginning, I knew nothing about dependent types and proof assistants. Therefore, we had to choose carefully what theorems to formalise. On one hand, the theorems should be non-trivial enough such that a substantial amount of work is required to be done. On the other hand, the theorems should not be too difficult because I am only a beginner in this area. Finally, we decided to go with the Automata Theory as it contains many components and I am familiar with the theory because its basic concepts were explained in the course *Model of Computation*.

1.2 Overview

Section 2 will be a brief introduction on Agda and dependent types. We will describe how Agda can be used as a proof assistant by formalising several small proofs. Experienced Agda users can skip this section and start from section 3 directly. In section 3, we will describe several researches that are also related to the formalisation of Automata Theory. Following the background, section 4 will be a detail description of our work. We will walk through the two components in our Agda formalisation. Note that the definitions, theorems and proofs written in this section are extracted from our Agda code. They may be different from their usual mathematical forms in order to adapt the environment in Type Theory. In section 5, we will discuss two possible extensions to our project: 1) Myhill-Nerode Theorem and 2) the Pumping Lemma. After that, in section 6, we will evaluate the project as a whole. Finally, the conclusions will be drawn.

2 Agda

Agda is a dependently-typed functional programming language and a proof assistant based on Intuitionistic Type Theory [12]. The current version (Agda 2) is rewritten by Norell [15] during his doctorate study at the Chalmers University of Technology. In this section, we will describe the basic features of Agda and how dependent types are employed to construct programs and proofs. Most of the materials presented below can also be found in the two tutorial papers [4] and [16]. Interested readers can read the two papers in order to get a more precise idea on how to work with Agda. Now, we will begin by showing how to do ordinary functional programming in Agda.

2.1 Simply Typed Functional Programming

Haskell is the implementation language of Agda, as shown below, Agda has borrowed many features from Haskell. In the following paragraphs, we will demonstrate how to define basic data types and functions.

Boolean We first declare the type of Boolean values in Agda.

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

Bool has two constructors: ***true*** and ***false***. These two constructors are also elements of ***Bool*** as they take no arguments. On the other hand, ***Bool*** itself is a member of the type ***Set***. The type of ***Set*** is $\boldsymbol{Set_1}$ and the type of $\boldsymbol{Set_1}$ is $\boldsymbol{Set_2}$. The type hierarchy goes on and becomes infinite. Now, let us define the negation of Boolean values.

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

Unlike in Haskell, a type signature must be provided explicitly for every function and all possible cases must be included in the function body while pattern matching. For instance, the function below will be rejected by the Agda compiler as the case **(*not false*)** is missing.

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

Natural Number Now, let us declare the type of natural numbers in Peano style.

```
data ℕ : Set where
```

```
zero  : ℕ
suc   : ℕ → ℕ
```

The constructor ***suc*** represents the successor of a given natural number. For instance, the number **1** is equivalent to (***suc zero***). Now, let us define the addition of natural numbers recursively as follow:

```
_+_  : 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 the type parameter a. The analogous data type in Agda is defined as follow:

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

Let us try to define a function which takes a list as the argument and returns the first element of the list.

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

What should be returned in case [ ]? In Haskell, the [ ] case can simply be skipped and an error will be produced by the compiler. However, as we have mentioned before, the function body must contains all the possible cases. One possible workaround is to return ***nothing*** of the ***Maybe*** type for case [ ]. Another solution is to constrain the arguments using dependent types such that the input list will always have at least one element.

2.2 Dependent Types

A dependent type is a type that depends on values of other types. For example, A^n is a vector that contains n elements of A. These kind of types is not possible to be declared in simply-typed systems like Haskell[1]and Ocaml. Now, let us look at how it is declared in Agda.

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

[1]Haskell itself does not support dependent types by its own. However, there are several APIs in Haskell that simulates dependent types, for example, Ivor [10] and GADT.

In the type signature, (***A* : *Set***) is the type parameter while $\mathbb{N} \rightarrow$ ***Set*** means that ***Vec*** takes a number ***n*** from $\mathbb{N}$ and produces a type that depends on ***n***. Different types will be produced by giving different natural numbers to the inductive family ***Vec***. For example, ***Vec A zero*** is the type of empty vectors and ***Vec A* 10** is another vector type with length ten.

Dependent types allow us to be more expressive and precise over type declaration. Let us declare the ***head*** function for ***Vec***.

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

Only the (***x* :: *xs***) case needs to be pattern matched because the type ***Vec A* (*suc n*)** ensures that the argument will never be [ ]. Apart from vectors, a type of binary search tree can also be declared in which any tree of this type is guaranteed to be sorted. However, this is not our major concern and thus we will not be looking into it. Interested readers can take a look at Section 6 in [4]. Furthermore, dependent types also allow us to encode predicate logic and program specifications as types. These two applications will be describe in later part and now, we will first discuss the idea of propositions as types.

2.3 Propositions as Types

In the 1930s, Curry identified the correspondence between propositions in propositional logic and types [5]. After that, in the 1960s, de Bruijn and Howard extended Curry's correspondence to predicate logic by introducing dependent types [6, 9]. Later on, Martin-Löf published his work, Intuitionistic Type Theory [12], which turned the correspondence into a new foundational system for constructive mathematics.

In the paragraphs below, we will show how the correspondence is formalised in Agda. Note that Intuitionistic Type Theory is based on constructive logic but not classical logic and there is a fundamental difference between them. Interested readers can take a look at [3]. Now, we will begin with propositional logic.

2.3.1 Propositional Logic

In general, Curry's correspondence states that a proposition can be interpreted as a set of its proofs. A proposition is true if and only if its set of proofs is inhabited, i.e. there is at least one element in the set; it is false if and only if its set of proofs is empty.

Truth For a proposition to be always true, its corresponding type must have at least one element.

```
data ⊤ : Set where
  tt : ⊤
```

Falsehood The proposition that is always false corresponds to a type having no elements at all.

```
data ⊥ : Set where
```

Conjunction Suppose ***A*** and ***B*** are propositions, then the proofs of their conjunction $\boldsymbol{A} \wedge \boldsymbol{B}$ should contain both a proof of ***A*** and a proof of ***B***. In Type Theory, it corresponds to the product type.

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

The above construction resembles the introduction rule of conjunction while the elimination rules are formalised as follow:

```
fst : {A B : Set} → A × B → A
fst (a , b) = a

snd : {A B : Set} → A × B → B
snd (a , b) = b
```

Disjunction Suppose ***A*** and ***B*** are propositions, then the proofs of their disjunction $\boldsymbol{A} \vee \boldsymbol{B}$ should contains either a proof of ***A*** or a proof of ***B***. In Type Theory, it is represented by the sum type.

```
data _⊎_ (A B : Set) : Set where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

The elimination rule of disjunction is defined as follow:

```
⊎-elim : {A B C : Set}
         → A ⊎ B
         → (A → C)
         → (B → C)
         → C
⊎-elim (inj₁ a) f g = f a
⊎-elim (inj₂ b) f g = g b
```

Negation Suppose ***A*** is a proposition, then its negation is defined as a function that transforms any arbitrary proof of ***A*** into the falsehood (⊥).

```
¬ : Set → Set
¬ A = A → ⊥
```

Implication We say that ***A*** implies ***B*** if and only if every proof of ***A*** can be transformed into a proof of ***B***. In Type Theory, it corresponds to a function from ***A*** to ***B***, i.e. ***A* → *B***.

Equivalence Two propositions ***A*** and ***B*** are equivalent if and only if ***A*** implies ***B*** and ***B*** implies ***A***. It can be considered as a conjunction of the two implications.

```
_⟺_ : Set → Set → Set
A ⟺ B = (A → B) × (B → A)
```

Now, by using the above constructions, we can formalise theorems of propositional logic in Agda. For example, we can prove that if ***P*** implies ***Q*** and ***Q*** implies ***R***, then ***P*** implies ***R***.

```
prop-lem : {P Q : Set}
         → (P → Q)
         → (Q → R)
         → (P → R)
prop-lem f g = λ p → g (f p)
```

By completing the function, we have provided an element to the type **(*P* → *Q*) → (*Q* → *R*) → (*P* → *R*)** and thus, we have also proved the theorem to be true.

2.3.2 Predicate Logic

We will now move on to predicate logic and introduce the universal (∀) and existential (∃) quantifiers. A predicate is represented by a dependent type in the form of ***A* → *Set***. For example, we can define the predicate of even numbers and odd numbers inductively as follow:

```
mutual
  data _isEven : ℕ → Set where
    base : zero isEven
    step : ∀ n → n isOdd → (suc n) isEven

  data _isOdd : ℕ → Set where
    step : ∀ n → n isEven → (suc n) isOdd
```

Universal Quantifier The interpretation of the universal quantifier is similar to implication. In order for $\forall \boldsymbol{x} \in \boldsymbol{A}.\ \boldsymbol{B}(\boldsymbol{x})$ to be true, every proof $(\boldsymbol{a})$ of $\boldsymbol{A}$ must be transformed into a proof of the predicate $\boldsymbol{B}[\boldsymbol{x} := \boldsymbol{a}]$. In Type Theory, it is represented by the function $(\boldsymbol{x} : \boldsymbol{A}) \rightarrow \boldsymbol{B}\ \boldsymbol{x}$. For example, we can prove by induction that for every natural number, it is either even or odd.

```
lem₁ : ∀ n → n isEven ⊎ n isOdd
lem₁ zero = inj₁ base
lem₁ (suc n) with lem₁ n
... | inj₁ nIsEven = inj₂ (step n nIsEven)
... | inj₂ nIsOdd = inj₁ (step n nIsOdd)
```

Existential Quantifier The interpretation of the existential quantifier is similar to conjunction. In order for $\exists \boldsymbol{x} \in \boldsymbol{A}.\ \boldsymbol{B}(\boldsymbol{x})$ to be true, a proof $(\boldsymbol{a})$ of $\boldsymbol{A}$ and a proof $(\boldsymbol{p})$ of the predicate $\boldsymbol{B}[\boldsymbol{x} := \boldsymbol{a}]$ must be provided. In Type Theory, it is represented by the generalised product type $\boldsymbol{\Sigma}$.

```
data Σ (A : Set) (B : A → Set) : where
  _,_ : (a : A) → B a → Σ A B
```

For simplicity, we will change the syntax of $\boldsymbol{\Sigma}$ to $\exists[\boldsymbol{x} \in \boldsymbol{A}]\ \boldsymbol{B}$. As an example, we can prove that there exists a natural number which is even.

```
lem₂ : ∃[ n ∈ ℕ ] (n isEven)
lem₂ = zero , base
```

2.3.3 Decidability

A proposition $\boldsymbol{P}$ is decidable if and only if there exists an algorithm that can decide whether it is true or false. It is defined as follow:

```
data Dec (A : Set) : Set where
  yes : A → Dec A
  no  : ¬ A → Dec A
```

For example, we can prove that the predicate of even numbers is decidable. Interested readers can try and complete the proofs.

```
lem₃ : ∀ n → ¬ (n isEven × n isOdd)
lem₃ = ?

lem₄ : ∀ n → n isEven → ¬ ((suc n) isEven)
lem₄ = ?
```

```
lem5 : ∀ n → ¬ (n isEven) → (suc n) isEven
lem5 = ?

even−dec : ∀ n → Dec (n isEven)
even−dec zero = yes base
even−dec (suc n) with even−dec n
... | yes nIsEven = no (lem4 n nIsEven)
... | no ¬nIsEven = yes (lem5 n ¬nIsEven)
```

2.3.4 Propositional Equality

One important feature of Type Theory is that the equality of propositions can also be defined as types. The equality relation is interpreted as follow:

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

This states that for any $\boldsymbol{x}$ in $\boldsymbol{A}$, $\boldsymbol{refl}$ is an element of the type $\boldsymbol{x \equiv x}$. More generally, $\boldsymbol{refl}$ is a proof of $\boldsymbol{x \equiv x'}$ provided that $\boldsymbol{x}$ and $\boldsymbol{x'}$ is the same after normalisation. For example, we can prove that $\exists \boldsymbol{n} \in \mathbb{N}.\ \boldsymbol{n = 1 + 1}$ as follow:

```
lem3 : ∃[ n ∈ ℕ ] n ≡ (1 + 1)
lem3 = suc (suc zero) , refl
```

We can put $\boldsymbol{refl}$ in the proof only because both $\boldsymbol{suc\ (suc\ zero)}$ and $\boldsymbol{1 + 1}$ have the same form after normalisation. Now, let us define the elimination rule of equality. The rule should allow us to substitute equivalence objects into any proposition.

```
subst : {A : Set}{x y : A} → (P : A → Set) → x ≡ y → P x → P
subst P refl p = p
```

We can also prove the congruency of equality.

```
cong : {A B : Set}{x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl
```

2.4 Program Specifications as Types

As we have mentioned before, dependent types also allow us to encode program specifications within the same platform. In order to demonstrate the idea, we will use the insertion function of sorted lists as an example. Let us begin by defining a predicate of sorted list (in ascending order).

```
All-lt : ℕ → List ℕ → Set
All-lt n [] = ⊤
All-lt n (x :: xs) = n ≤ x × All-lt n xs

Sorted-ASC : List ℕ → Set
Sorted-ASC [] = ⊤
Sorted-ASC (x :: xs) = All-lt x xs × Sorted-ASC xs
```

For simplicity, only the list of natural numbers is considered. Note that *All-lt* defines the condition where a given number is smaller than all the numbers inside a given list. Now, let us define an insertion function that takes a natural number and a list as the arguments and returns a list of natural numbers. The insertion function is designed in a way that if the input list is already sorted, then the output list will also be sorted.

```
insert : ℕ → List ℕ → List ℕ
insert n [] = n :: []
insert n (x :: xs) with n ≤? x
... | yes _ = n :: (x :: xs)
... | no  _ = x :: insert n xs
```

Note that _ ≤?_ has the type ∀ ***n m → Dec (n ≤ m)***. It is a proof of the decidability of _ ≤ _ and it can also be used to determine whether a given number ***n*** is less than or equal to another number ***m***. Now, let us encode the specification of the insertion function as follow:

```
insert-sorted : ∀ {n} {as}
                → Sorted-ASC as
                → Sorted-ASC (insert n as)
```

In the type signature, (*Sorted-ASC as*) corresponds to the pre-condition and (*Sorted-ASC* (*insert n as*)) corresponds to the post-condition. Once we have completed the function, we will also have proved the specification to be true. Interested readers are recommended to finished the proof.

3 Related Work

3.1 Regular Expressions in Agda

Agular and Mannaa published a similar work [1] in 2009. They constructed a decider for regular expressions which can determine whether a given string is accepted by a given regular expression. The decider was based on the calculation of the derivation of a regular expression which only needs to convert the regular expression into part of an automaton. Their decider was implemented using the $\boldsymbol{Maybe}$ type as follow:

```
accept : (re : RegExp) → (as : List carrier) → Maybe (as ∈‿ ⟦re
```

When a string is accepted by the regular expression, i.e. $\boldsymbol{w \in L(e)}$, the decider will return its proof. However it fails to generate a proof for the opposite case, i.e. $\boldsymbol{w \notin L(e)}$. As they explained in the paper, it is not possible without converting the regular expression into the entire finite automaton.

3.2 Certified Parsing of Regular Languages in Agda

While in 2013, Firsov and Uustalu also published another related research paper [7]. They translated regular expressions into NFA and proved that their accepting languages are equal. Unlike Agular and Mannaa's decider, Firsov and Uustalu's algorithm could generate proofs for both cases. In their definition of NFA, the set of states $\boldsymbol{Q}$ and its subsets are represented as vectors while the transition function $\boldsymbol{\delta}$ takes an alphabet as the argument and returns a matrix representation of the transition table.

```
record NFA : Set where
  field
    |Q| : ℕ
    δ   : Σ → |Q| * |Q|
    I   : 1 * |Q|
    F   : |Q| * 1
```

Note that _ ∗ _ is an inductive family that takes two natural numbers $\boldsymbol{n}$ and $\boldsymbol{m}$ and produces a matrix type $\boldsymbol{n \times m}$. This representation allows us to iterate the set easily but it looks unnatural compare to the actual mathematical definition of NFA.

4 Formalisation in Type Theory

Let us recall the two components of the formalisation: 1) translating any regular expressions to a DFA and 2) proving the correctness of the translation.

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 proos 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 the following parts, we will walk through the formalisation of each of the above steps together with their correctness proofs. Note that all the definitions, theorems, lemmas and proofs wriiten in below are adapted to the formalisation in Agda. Now, before we go into regular expressions and automata, we first need to have a representation of subsets and languages as they are fundamental elements in the theory.

4.1 Subsets and Decidable Subsets

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

When declaring a subset in Agda, we can write $SubA = \lambda\ a \to ?$, 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 $\lambda\ a \to P\ a$. 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 $DecSubset\ A = A \to Bool$. 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.

4.1.1 Operations on Subsets

4.1.2 Operations on Decidable Subsets

Now, by using the representation of subset, we can define languages, regular expressions and finite automata.

4.2 Languages

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

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. $\Sigma^* = List\ \Sigma$.

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, $Language = Subset\ \Sigma^*$. Notice that $Subset$ instead of $DecSubset$ is used because not every language is decidable.

4.2.1 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 $L_1 \cup L_2 = \lambda\ w \to w \in L_1\ \uplus\ w \in L_2$.

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 $L_1 \bullet L_2 = \lambda\ w \to \exists[u \in \Sigma^*]\ \exists[v \in \Sigma^*](u \in L_1 \times v \in L_2 \times w \equiv u\ +\!+\ v)$.

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

```
_ˆ_ : Language → Language → Language
L ˆ zero    = ⟦ϵ⟧
L ˆ (suc n) = L • L ˆ n
```

4.3 Regular Languages and Regular Expressions

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 ≈ ø → 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
  Ø    : RegExp
  ϵ    : RegExp
  σ    : Σ → RegExp
  _|_  : RegExp → RegExp → RegExp
  _·_  : RegExp → RegExp → RegExp
  _*  : RegExp → RegExp
```

Listing 3: Languages denoted by regular expressions

```
L^R : RegExp → Language
L^R Ø = ø
L^R ϵ = ⟦ϵ⟧
L^R (σ a) = ⟦a⟧
L^R (e₁ | e₂) = L^R e₁ ∪ L^R e₂
L^R (e₁ · e₂) = L^R e₁ • L^R e₂
L^R (e*) = (L^R e) ⋆
```

4.4 ϵ-Non-deterministic Finite Automata

By now, the set of strings we have considered are in the form of *List* Σ^*. 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:

```
data Σ^e : Set where
  α : Σ → Σ^e
  E : Σ^e
```

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 ϵ-NFA : Set₁ where
  field
    Q       : Set
    δ       : Q → Σᵉ → DecSubset Q
    q₀      : Q
    F       : DecSubset Q
    ∀qEq    : ∀ q → q ∈ᵈ δ q E
    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 Σ : 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 $a \in \Sigma^e$.

```
_⊢_ : (Q × Σ^e × Σ^e*) → (Q × Σ^e*) → Set
(q , a , w) ⊢ (q' , w') = w ≡ w' × q' ∈^d δ q a
```

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 $C_1, C_2, ..., C_{k-1}$ such that $C_i \vdash C_{i+1}$ for all $0 \leq i \leq k$.

```
_⊢^k_-_ : (Q × Σ^e*) → ℕ → (Q × Σ^e*) → Set
(q , w^e) ⊢^k zero - (q' , w^e')
  = q ≡ q' × w^e ≡ w^e'
(q , w^e) ⊢^k suc n - (q' , w^e')
  = ∃[ p ∈ Q ] ∃[ a^e ∈ Σ^e ] ∃[ u^e ∈ Σ^e* ]
     (w^e ≡ a^e :: u^e × (q , a^e , u^e) ⊢ (p , u^e) × (p , u^e) ⊢^k n - (q' , w^e'))
```

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

```
_⊢*_ : (Q × Σ^e*) → (Q × Σ^e*) → Set
(q , w^e) ⊢* (q' , w^e') = ∃[ n ∈ ℕ ] (q , w^e) ⊢^k n - (q' , w^e')
```

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 $\{\ w \mid \exists w^e \in \Sigma^{e*}.\ w = to\Sigma^*(w^e) \wedge \exists q \in F.\ (q_0\ ,\ w^e) \vdash^* (q\ ,\ \epsilon)\ \}$.

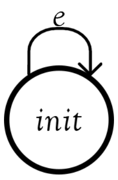
```
L^eN : ε-NFA → Language
L^eN nfa = λ w →
        ∃[ w^e ∈ Σ^e* ] (w ≡ toΣ* w^e × (∃[ q ∈ Q ] (q ∈^d F × (q₀ , w^e) ⊢* (q , []))))
```

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

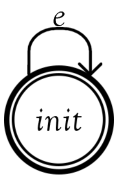
4.5 Thompson's Construction

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

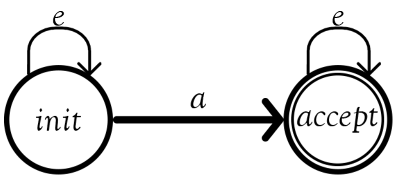
1. for $\emptyset$, we have $M = (\{init\},\ \Sigma^e,\ \delta,\ init,\ \emptyset)$ and graphically



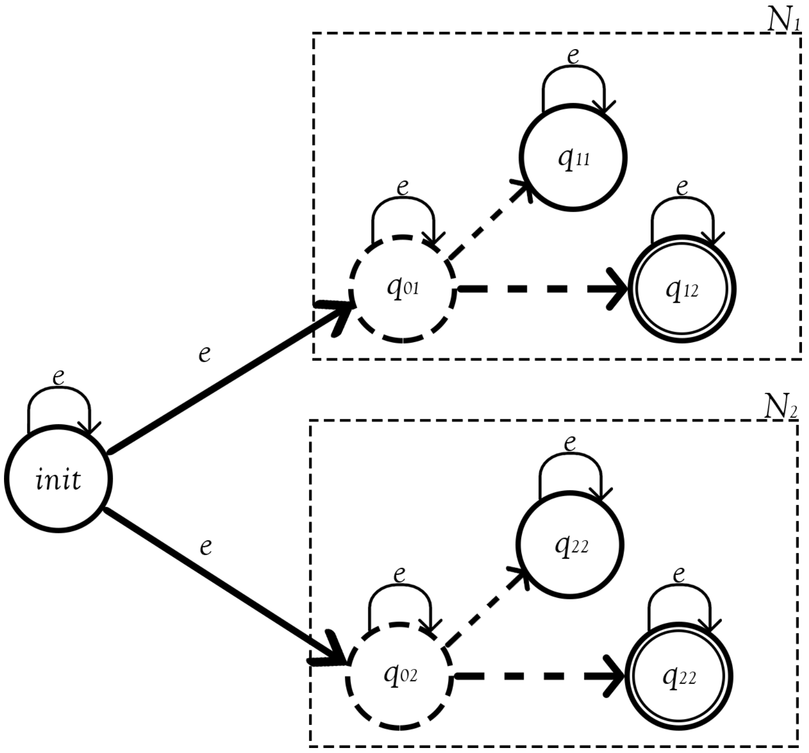
2. for ϵ, we have $M = (\{init\},\ \Sigma^e,\ \delta,\ init,\ \{init\})$ and graphically



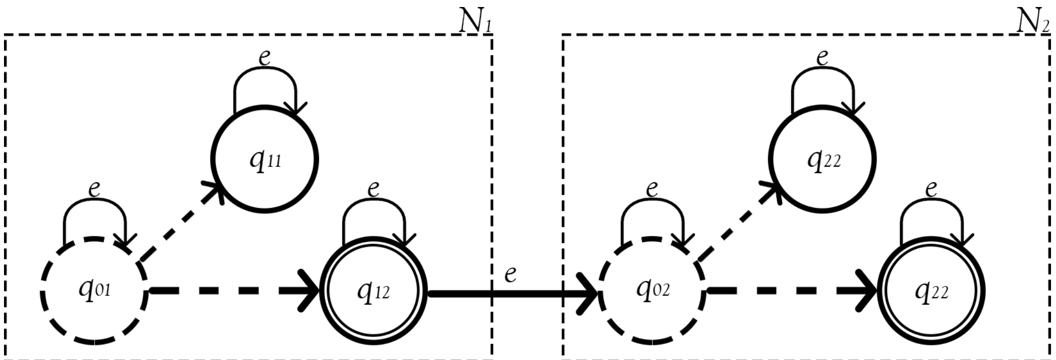
3. for a, we have $M = (\{init, accept\},\ \Sigma^e,\ \delta,\ init,\ \{accept\})$ and graphically



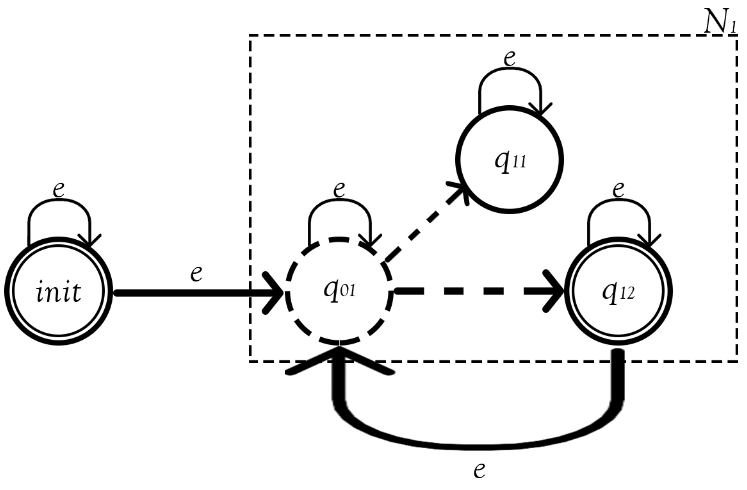
4. if $N_1 = (Q_1,\ \delta_1,\ q_{01},\ F_1)$ and $N_2 = (Q_2,\ \delta_2,\ q_{02},\ F_2)$ are ϵ-NFAs translated from the regular expressions e_1 and e_2 respectively, then
   (a) for $(e_1 \mid e_2)$, we have $M = (\{init\} \cup Q_1 \cup Q_2,\ \Sigma^e,\ \delta,\ init,\ F_1 \cup F_2)$ and graphically



(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. $L(e) = L(translted\ \epsilon\text{-NFA})$.

Proof 1.1 We have to prove that for any regular expressions e, $L(e) \subseteq L(translated\ \epsilon\text{-}NFA)$ and $L(e) \supseteq L(translated\ \epsilon\text{-}NFA)$ 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 $u^e \epsilon\epsilon v^e$; 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 $n = 0$, $L\ \hat{}\ 0 = ...$

3.2) if M accepts w, ...

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

4.6 Non-deterministic Finite Automata

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 Σ : *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 $a \in \Sigma$.

```
_⊢_ : (Q × Σ × Σ*) → (Q × Σ*) → Set
(q , a , w) ⊢ (q' , w') = w ≡ w' × q' ∈ᵈ δ q a
```

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 $C_1, C_2, ..., C_{k-1}$ such that $C_i \vdash C_{i+1}$ for all $0 \leq i \leq k$.

```
_⊢ᵏ_─_ : (Q × Σ*) → ℕ → (Q × Σ*) → Set
(q , w) ⊢ᵏ zero ─ (q' , w')
  = q ≡ q' × w ≡ w'
(q , w) ⊢ᵏ suc n ─ (q' , w')
  = ∃[ p ∈ Q ] ∃[ a ∈ Σ ] ∃[ u ∈ Σ* ]
     (w ≡ a :: u × (q , a , u) ⊢ (p , u) × (p , u) ⊢ᵏ n ─ (q' , w'))
```

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

```
_⊢*_ : (Q × Σ*) → (Q × Σ*) → Set
(q , w) ⊢* (q' , w') = ∃[ n ∈ ℕ ] (q , w) ⊢ᵏ n ─ (q' , w')
```

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 $\{ \ w \mid \exists q \in F. \ (q_0 \ , \ w) \vdash^* (q \ , \ \epsilon) \ \}$.

```
Lᴺ : NFA → Language
Lᴺ nfa = λ w → ∃[ q ∈ Q ] (q ∈ᵈ F × (q₀ , w) ⊢* (q , []))
```

4.7 Removing ϵ-transitions

...

4.8 Deterministic Finite Automata

...

4.9 Powerset Construction

...

4.10 Minimal DFA

...

4.11 Minimising DFA

...

5 Further Extensions

Myhill-Nerode Theorem, Pumping Lemma

6 Evaluation

In this section, we will evaluate 1) the Agda code based on the different representations of mathematical objects that we have chosen and 2) the development process. Furthermore, we will discuss the feasibility of formalising mathematics and programming logic in practice based on our own experiences.

6.1 Different choices of representations

In computer proofs, an abstract mathematical object usually requires a concrete representation. The consequence is that different representations will lead to different formalisations and thus contribute to the easiness or difficulty in completing the proofs. In the following paragraphs, we will discuss the decisions we have made and their effects on the project.

The set of states (Q) and its subsets As we have mentioned in section 3, Firsov and Uustalu [7] represent the set of states $\boldsymbol{Q}$ and its subsets as column matrices. However, this representation looks unnatural compare to the actual mathematical definition. Therefore, at the beginning, we intended to avoid the vector representation and to to represent the sets in an abstract form. In our approach, the set of states are represented as a data type in Agda, i.e. $\boldsymbol{Q : Set}$, and its subsets are represented as unary functions on $\boldsymbol{Q}$, i.e. $\boldsymbol{DecSubset\ Q = Q \to Bool}$.

Our definition allows us to finish the proofs in **Correctness/RegExp-eNFA.agda** without having to manipulate matrices. The proofs also look much more natural compare to that in [7]. However, the problem arises when the sets have to be iterated during the computation of the ϵ-closures because it is not possible to iterate the sets using this representation. Therefore, in the current version, several extra fields are included in the automata including $\boldsymbol{It}$ – a vector containing all the states in $\boldsymbol{Q}$. With $\boldsymbol{It}$, the subsets of $\boldsymbol{Q}$ can be iterated by applying their own functions $\boldsymbol{Q \to Bool}$ to all the elements in $\boldsymbol{It}$. Note that the vector $\boldsymbol{It}$ is equivalent to the vector representation of the set of states.

The languages accepted by regular expressions and finite automata At first, the accepting language of regular expressions was defined as a decidable subset, i.e. $\boldsymbol{L^R : RegExp \to DecSubset\ \Sigma^*}$. The decision was reasonable because the language has been proved decidable for many years. However, when we were defining the language, we were also constructing a boolean decider for regular expressions. This added a great amount of difficulties in writing the proofs because the proofs had to be built based on the decider. For example, in the concatenation case, an extra recursive function was needed to iterate different combinations of input string. Furthermore, the decidability of the language is not necessary in proving the equality. Therefore, in the current version, the language is defined as a general subset, i.e. $\boldsymbol{L^R : RegExp \to Subset\ \Sigma^*}$. This definition allows us to separate out the decider from the mathematical definition and to prove their equality in an abstract level.

The same situation also happened to the accepting language of finite automata. At first, the accepting language of NFA was obtained by running an algorithm that produces all the reachable states from $\boldsymbol{q_0}$ using the input string. The algorithm was also a decider for NFA. Once again, this definition mixes the decider and the proposition together and the decidability of NFA is not necessary in the equality. Therefore, in the current version, the accepting language of NFA is defined in an abstract form.

However, this does not mean that we can ignore the algorithms. In fact, the decidability of DFA requires an algorithm to return the last state after running the DFA. In this case, the correctness proof of the algorithm is required. However, compare to the algorithm used in running NFA, this algorithm is much simpler which also makes its correctness proof very easy to finish. Furthermore, the decidability of NFA and regular expressions follows directly after their accepting languages are proved to be equal.

The set of reachable states from q_0 Let us recall the definition of reachable states from $\boldsymbol{q_0}$.

```
-- Reachable from q₀
Reachable : Q → Set
Reachable q = Σ[ w ∈ Σ* ] (q₀ , w) ⊢* (q , [])

data Qᴿ : Set where
  reach : ∀ q → Reachable q → Qᴿ
```

We say that a state $\boldsymbol{q}$ is reachable if and only if there exists a string $\boldsymbol{w}$ that can take $\boldsymbol{q_0}$ to $\boldsymbol{q}$. Therefore, the set $\boldsymbol{Q^R}$ should contains all and only the reachable states in $\boldsymbol{Q}$. However, there may exist more than one reachability proof for a single state. This implies that there may be more than one element in $\boldsymbol{Q^R}$ having the same state in $\boldsymbol{Q}$. Therefore, $\boldsymbol{Q^R}$ may be larger than the original set $\boldsymbol{Q}$ or even worse, it may be infinite. This leads to a problem when a DFA is constructed using $\boldsymbol{Q^R}$ as the set of states. If $\boldsymbol{Q^R}$ is infinite, then there is no possible way to iterate the set during quotient construction. Even if the set $\boldsymbol{Q^R}$ is finite, it contradicts our original intention to minimise the set of states. This is also one of the reasons why our formalisation of quotient construction cannot be completed. However, this has no effects on the proof of $\boldsymbol{L(DFA) = L(MDFA)}$ because we can provide an equality relation of states of a DFA. In the translation from DFA to MDFA, we defined the equality relation as follow: any two states in $\boldsymbol{Q^R}$ are equal if and only if the input states are equal. Therefore, two elements with same state but different reachability proofs are still considered the same in the new DFA.

One possible way to solve the problem is to re-define the reachability of a state such that any reachable state will have a unique reachability proof. For example, the representative proof can be selected by choosing the shortest string $\boldsymbol{(w)}$ sorted according to alphabetical order. This also requires a proof that the chosen representative is unique. Another solution is to use homotopy type to declare the set $\boldsymbol{Q^R}$. This type allows us to group different reachability proofs into a single element such that every state will only appear once in $\boldsymbol{Q^R}$.

6.2 Development Process

As we have mentioned before, the parts related to quotient construction was not completed. One of the reasons has been discussed in the previous part. However, the major reason is that there was only very limited time left when we started the quotient construction. In the following paragraphs, we will evaluate the whole development process and discuss what could have been done better.

In the first 6 weeks, I was struggling to find the most suitable representations for regular expressions, finite automata and their accepting languages. During the time, I was rushing in coding without thinking about the whole picture of the theory. As a result, many bad decisions had been made, for example, omitting the ϵ-transitions in the translation process and trying to prove the decidability of regular expressions directly. After taking the advice from my supervisor, I followed the definitions in the book [2] and started writing a framework of the theory. After that, in just one week, the Agda codes that formed the basis of the final version were developed by using the framework. One could argue that the work done in the first 6 weeks also contribute to those Agda codes but there is no doubt the written framework highly influenced the development. Therefore, even though it is convenient to write proofs using the interactive features in Agda, it would still better to start with a framework in early stages.

After that, the development went smooth until the first week of the second semester. During the time, I started to prove several properties of ϵ-closures. The plan was to finish this part within 2 weeks. However, 4 weeks were spent on it and very little progress were made. These 4 weeks were crucial to the schedule and the time should have been spent more wisely on other parts of the project such as powerset construction and the report.

6.3 Computer-aided verification in practice

In order to evaluate the feasibility of performing computer-aided verification in practice, we will discuss:1) the difference between computer proofs and written proofs, 2) the easiness or difficulty in formalising mathematics and programming logic and 3) the difference between computer-aided verification and testing.

6.3.1 Computer proofs and written proofs

According to Geuvers [8], a proof has two major roles: 1) to convince the readers that a statement is correct and 2) to explain why a statement is correct. The first role requires the proof to be convincing. The second role requires the proof to be able to give an intuition of why the statement is correct.

Correctness Traditionally, when a mathematicians submit the proof of their concepts, a group of other mathematicians will evaluate and check the correctness of the proof. Alternatively, if the proof is formalised in a proof assistant, it will be checked automatically by the compiler. The only difference is that we are now relying on the compiler and the machine that runs the compile rather than the group of mathematicians. Therefore, if the compiler and the machine works properly, then any formalised proof that can be compiled without errors are said to be correct. Furthermore, a proof is consist of smaller reasoning steps. We can say that the a proof is correct if and only if all the reasoning steps within the proof are correct. When writing proofs in paper, the proofs of some obvious lemmas are usually omitted and this sometimes leads to mistakes. However, in most proof assistants, the proofs of every lemma must be provided explicitly. Therefore, the correctness of a computer proof always depends on the correctness of the smaller reasoning steps within it.

Readability The second purpose of a proof is to explain why a certain statement is correct. Let us consider the following code snippet extracted from **Correctness/RegExp-eNFA.agda**.

```
lem₃ : ∀ wᵉ n q₁
       → (q₀ , wᵉ)  ⊢ᵏ suc n ─ (inj q₁ , [])
       → Σ[ n₁ ∈ ℕ ] Σ[ uᵉ ∈ Σᵉ* ] (toΣ* wᵉ ≡ toΣ* uᵉ × (inj q₀₁ , uᵉ) ⊢ᵏ n₁ ─ (inj q₁ , []))
lem₃ ._ zero    q₁  (inj .q₁  , α _ , .[] , refl , (refl ,   ()) , (refl , refl))
lem₃ ._ zero    q₁  (inj .q₁  , E   , .[] , refl , (refl , prf₁) , (refl , refl)) with Q₁? q₁ q₀₁
lem₃ ._ zero   .q₀₁ (inj .q₀₁ , E   , .[] , refl , (refl , prf₁) , (refl , refl)) | yes refl  = zero , [] , (refl , (refl , refl))
lem₃ ._ zero    q₁  (inj .q₁  , E   , .[] , refl , (refl ,   ()) , (refl , refl)) | no  p≢q₀₁
lem₃ ._ (suc n) q₁  (init     , α _ ,  uᵉ , refl , (refl ,   ()) , prf₂)
lem₃ ._ (suc n) q₁  (init     , E   ,  uᵉ , refl , (refl , prf₁) , prf₂) = lem₃ uᵉ n q₁ prf₂
lem₃ ._ (suc n) q₁  (inj  p   , α _ ,  uᵉ , refl , (refl ,   ()) , prf₂)
lem₃ ._ (suc n) q₁  (inj  p   , E   ,  uᵉ , refl , (refl , prf₁) , prf₂) with Q₁? p q₀₁
lem₃ ._ (suc n) q₁  (inj .q₀₁ , E   ,  uᵉ , refl , (refl , prf₁) , prf₂) | yes refl  = suc n , uᵉ , refl , prf₂
lem₃ ._ (suc n) q₁  (inj  p   , E   ,  uᵉ , refl , (refl ,   ()) , prf₂) | no  p≢q₀₁
```

The above code is a proof that if $\boldsymbol{w^e}$ can take $\boldsymbol{q_0}$ to another state $\boldsymbol{inj\ q_1}$ in an ϵ-NFA translated from a regular expression $\boldsymbol{e^*}$, then there exists a number $\boldsymbol{n_1}$ and an ϵ-string $\boldsymbol{u^e}$ that will take $\boldsymbol{inj\ q_{01}}$ to $\boldsymbol{inj\ q_1}$ where $\boldsymbol{q_{01}}$ is the start state of $\boldsymbol{e}$ and $\boldsymbol{q_1}$ is a state in $\boldsymbol{e}$. There are several techniques used in the proof including the induction on natural numbers and case analysis on state comparison. However, by just looking at the function body, the proving process can hardly be understood. Therefore, in general, a computer proof is very inadequate on this purpose and thus an outline of the proof in natural language is still necessary.

Although computer proof seems to be unreadable, the advantages are still impressive. For example, a group of mathematicians may need months to validate a very long piece of proof, but a computer proof may only need days to compile.

6.3.2 Easiness or difficulty in the process of formalisation

As a computer science student without a strong mathematical background, I find it not very difficult to do the formalisation. There are many similarities between writing codes and writing proofs, for example, pattern matching and case analysis, recursive function and mathematical induction. Furthermore, Agda is convenient to use for its interactive features. The features allow us to easily know what is happening inside the proof body by showing the goal and all the elements in the proof. Also, many theorems can be automatically proved by Agda.

On the other hand, most of the proof assistants based on dependent types support only primitive or structural recursion. Many algorithms can be very difficult to implement under this limitation. For example, the usual algorithm that is used to compute ϵ-closure is as follow:

```
Suppose we are finding the $\epsilon$-closure of a state $q$:
1) Let $A$ be a set of states
2) Put $q$ in the set $A$
3) Loop until there is no new elements add to $A$
   3.1) For every state $p$ in $A$, if another state $r$ is reachable
        from $p$ with one $\epsilon$-transition, we put it in $A$
4) The resulting set $A$ is the $\epsilon$-closure of $q$
```

This kind of algorithms cannot be implemented directly without modifications. Furthermore, most of the proof assistants are not Turing-complete which means that there might be some useful algorithms which cannot be expressed.

6.3.3 Computer-aided verification and testing

The most common way to verify a program is via testing. However, no matter how sophisticated the design of the test cases is, the program still cannot be proved to be 100% correct. On the other hand, total correctness can be achieved by proving the specifications of a program. Already in 1997, Necula [13] has raised the notion of *Proof Carrying code.* The idea is to accompany program with several proofs that proves the program specifications within within the same platform. In fact, there is already an extraction mechanism [11] in Coq that allows us to extract proofs and functions in Coq into Ocaml, Haskell or Scheme programs.

7 Conclusion

Let us recall the two components of our

Bibliography

[1] Alexandre Agular and Bassel Mannaa. Regular expressions in agda, 2009.

[2] Alfred V. Aho and Jeffery V. Ullman. The theory of parsing, translation and compiling. volume i: Parsing, 1972.

[3] Jeremy Avigad. Classical and constructive logic, 2000.

[4] Ana Bove and Peter Dybjer. Dependent types at work. In *LERNET 2008. LNCS*, pages 57–99. Springer, 2009.

[5] Haskell Curry. Functionality in combinatory logic, 1934.

[6] Nicolaas de Bruijn. Automath, a language for mathematics, 1968.

[7] Denis Firsov and Tarmo Uustalu. Certified parsing of regular languages, 2013.

[8] Herman Geuvers. Proof assistants: history, ideas and future, 2009.

[9] William A. Howard. The formulæ-as-types notion of construction. In Philippe De Groote, editor, *The Curry-Howard Isomorphism*. Academia, 1995.

[10] Ivor. `https://eb.host.cs.st-andrews.ac.uk/ivor.php`. Accessed: 28th March 2016.

[11] Pierre Letouzey. Extraction in coq: An overview. In *Proceedings of the 4th Conference on Computability in Europe: Logic and Theory of Algorithms*, CiE '08, pages 359–369, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Per Martin-Löf. Intuitionistic type theory, 1984.

[13] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[14] Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Clarendon Press, 1990.

[15] Ulf Norell. Towards a practical programming language based on dependent type theory, 2007.

[16] Ulf Norell and James Chapman. Dependently typed programming in agda. In *Lecture Notes from the Summer School in Advanced Functional Programming*, 2008.

[17] The agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.HomePage`. Accessed: 12th March 2016.

[18] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.

[19] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015.

Appendices

Agda Code?