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

ϵ-NFA	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 [13]. ... [small intro to type theory → type theory and proof assistant and dependent types → Agda will be used]

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 consists 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 was new to 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 its basic concepts were explained in the course *Model of Computation*.

1.2 Outline

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 giving examples of formalised 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 of 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 to the environment of 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 [13]. The current version (Agda 2) is rewritten by Norell [16] 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 [17]. Interested readers can read the two papers in order to get a more precise idea on how to work with Agda. 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 and 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. Let us begin with the boolean data type.

Boolean In Haskell, the boolean type can be defined as $data\ Bool = True \mid False$. While in Agda, the syntax of the declaration is slightly different.

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

Bool has two constructors: *true* and $false$. These two constructors are also the elements of *Bool* as they take no arguments. Furthermore, *Bool* itself is a member of the type *Set*. The type of *Set* is Set_1 and the type of Set_1 is 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. Furthermore, all possible cases must be pattern matched in the function body. For instance, the function below will be rejected by the Agda compiler as the case ($false$) is missing.

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

Natural Number We will define 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
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
```

Now, let us try to define a function that will return the first element of a given list.

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

What should be returned for case [ ]? In Haskell, the [ ] case can simply be ignored and an error will be produced by the compiler. However, as we have mentioned before, all possible cases must be pattern matched in an Agda function. One possible workaround is to return *nothing* – an element of the *Maybe* type. Another solution is to constrain the argument 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. It is not possible to declare these kinds of types 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 [11] and GADT.

In the type signature, $(\mathbb{N} \to Set)$ means that Vec takes a number n from $\mathbb{N}$ and produces a type that depends on n. The inductive family Vec will produce different types with different natural numbers. 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 define 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 an element of the type $(Vec\ A\ (suc\ n))$ must be in the form of $(x :: xs)$. 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. 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 described in later part after we have discussed 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, 10]. Later on, Martin-Löf published his work, Intuitionistic Type Theory [13], 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 by showing how propositional logic is formalised in Agda.

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. Let us begin with the formalisation of $Truth$ – the proposition that is always true.

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 a proof of their conjunction, $A \wedge B$, 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. 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 a proof of their disjunction, $A \vee B$, should contain 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 ($\bot$).

```
¬ : 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. $A \to B$.

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 in propositional logic. For example, we can prove that if P implies Q and Q implies R, then P implies R. The corresponding proof in Agda is as follow:

```
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 $(P \to Q) \to (Q \to R) \to (P \to R)$ 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 ($\forall$) and existential ($\exists$) quantifiers. Suppose A is a set, then a predicate on A corresponds to a dependent type in the form of $A \to Set$. For example, we can define the predicates of even numbers and odd numbers inductively as follow:

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

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

Universal Quantifier The interpretation of the universal quantifier is similar to that of implication. In order for $\forall x \in A.\ B(x)$ to be true, every proof x of A must be transformed into a proof of the predicate $B[a := x]$. In Type Theory, it is represented by the function $(x : A) \to B\ x$. 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 (even n nIsEven)
... | inj₂ nIsOdd = inj₁ (odd n nIsOdd)
```

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

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

For simplicity, we will change the syntax of Σ type to $\exists[x \in A]\ B\ x$. As an example, let us prove that there exists an even natural number.

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

2.3.3 Decidability

A proposition is decidable if and only if there exists an algorithm that can decide whether the proposition is true or false. It is defined in Agda 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 following 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 of the important features of Type Theory is to encode the equality relation of propositions 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 x in A, $refl$ is an element of the type $x \equiv x$. More generally, $refl$ is a proof of $x \equiv x'$ provided that x and x' is the same after normalisation. For example, we can prove that $suc\ (suc\ zero) = 1 + 1$ as follow:

```
lem₃ : suc (suc zero) ≡ (1 + 1)
lem₃ = refl
```

We can put $refl$ in the proof only because both $suc\ (suc\ zero)$ and $1 + 1$ 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 give an example on the insertion function of sorted lists. Let us begin by defining a predicate of sorted list (in ascending order). For simplicity, we only consider the list of natural numbers.

```
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
```

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 $\leq?$ has the type $\forall\ n\ m \rightarrow Dec\ (n \leq m)$. It is a proof of the decidability of $\leq$ 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 have also proved the specification to be true. Readers are recommended to finish the proof.

3 Related Work

In this section, we will describe some researches that also aim to formalise Automata Theory in Agda. Apart from using Agda, there are many other researches that do the formalisation in other proof assistant, For example, Doczkal et al. [7] translated regular expressions into minimal DFA and proved the correctness of the translation in Coq. However, we will only focus on the researches that used the same formalisation platform.

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 implemented by converting a regular expression into a partial automaton. Here is the type signature of the decider:

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

accept takes a regular expression re and a list of alphabets as as the arguments. If the string as is accepted by the regular expression re, i.e. $as \in L(re)$, the decider will return its proof. However it fails to generate a proof for the opposite case, i.e. $as \notin L(re)$. As they explained in the paper, it is not possible without converting the regular expression into the entire automaton.

3.2 Certified Parsing of Regular Languages in Agda

While in 2013, Firsov and Uustalu published another related research paper [8]. 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 can generate proofs for both cases. In their definition of NFA, the set of states (Q) and its subsets are represented as vectors; and the transition function (δ) 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 $\_ * \_$ here is an inductive family that takes two natural numbers n and m and produces a matrix type of $n \times m$. This representation of sets allows us to iterate the sets 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 our formalisation: the translation of regular expressions to DFA and the correctness proofs of the translation. The translation is divided into several steps. Firstly, any regular expression is converted into an ϵ-NFA using Thompson's construction [19]. Secondly, all the ϵ-transitions are removed by computing the ϵ-closures. Thirdly, a DFA is built by using powerset construction. After that, all the unreachable states are removed. Finally, a minimal DFA is obtained by using quotient construction. The translation is correct if and only if 1) the accepting languages of the regular expression and its translated output are equal, i.e. $L(regex) = L(\text{translated } \epsilon\text{-NFA}) = L(\text{translated DFA}) = L(\text{translated MDFA})$ and 2) the translated MDFA is minimal.

In this section, we will walk through the formalisation of each of the above steps and their correctness proofs. Note that all the definitions, theorems, lemmas and proofs written in this section are adapted to the environment of Agda. Now let us begin with the representation of subsets and languages.

4.1 Subsets and Decidable Subsets

The types of subsets and decidable subsets are defined in **Subset.agda** and **Subset/Decidable-Subset.agda** respectively along with their operations such as membership ($\in$), subset ($\subseteq$), superset ($\supseteq$) and equality ($=$). Let us begin with the definition of general subsets. To separate the operations of subsets and decidable subsets, all the operations of decidable subset are denoted by the superscript (^{d}), e.g. $\in^d$ is the membership decider for decidable subsets.

Definition 1. Suppose A is a set, then its subsets are represented as unary functions on A in Type Theory, i.e. $Subset\ A = A \to Set$.

In our definition, a subset is a function from A to Set. When declaring a subset, we can write $sub = \lambda\ (x : A) \to\ P\ x$. $P\ x$ defines the conditions for x to be included in sub. This construction is very similar to set comprehension. For example, the above subset corresponds to $\{x \mid x \in A,\ P(x)\}$. Furthermore, sub is also a predicate on A as its type is in the form of $A \to Set$ and its decidability will remain unknown until it is either proved or disproved.

Definition 2. Another representation of subset is $DecSubset\ A = A \to Bool$. Unlike $Subset$, its decidability is ensured by its definition.

The two representations have different roles in the project. For example, *Language* is defined using $Subset$ as not every language is decidable. For other parts in the project such as the subsets of states in an automaton, $DecSubset$ is used because the decidability is assumed.

4.2 Languages

The type of languages is defined in **Language.agda** along with its operations and lemmas such as union ($\cup$), concatenation ($\bullet$) and closure ($\star$).

We represent the set of alphabets Σ as a data type in Type Theory, i.e. $\Sigma : Set$. Note that the equality relation of Σ needs to be decidable. In Agda, they are passed to every module as parameters, e.g. $module\ Language(\Sigma : Set)\ (dec : DecEq\ \Sigma)\ where$.

Definition 3. We first define Σ^* as the set of all strings over Σ. In our approach, it is expressed as a list of alphabets, i.e. $\Sigma^* = List\ \Sigma$.

For example, $(A :: g :: d :: a :: [\ ])$ is equivalent to the string 'Agda' and the empty list $[\ ]$ represents the empty string (ϵ). In this way, the first alphabet can be extracted from the input string by pattern matching in order to run a transition from a particular state to another state in an automaton.

Definition 4. A language is defined as a subset of Σ^*, i.e. $Language = Subset\ \Sigma^*$. Note that $Subset$ instead of $DecSubset$ is used because not every language is decidable.

4.2.1 Operations on Languages

Definition 5. Suppose L_1 and L_2 are languages, then the union of the two languages, $L_1 \cup L_2$, is given by the set $\{w \mid w \in L_1 \vee w \in L_2\}$. In Type Theory, we have $L_1 \cup L_2 = \lambda\ w \to w \in L_1 \uplus w \in L_2$.

Definition 6. Suppose L_1 and L_2 are languages, then the concatenation of the two languages, $L_1 \bullet L_2$, is given by the set $\{w \mid \exists u \in L_1.\ \exists v \in L_2.\ w = uv\}$. In Type Theory, we have $L_1 \bullet L_2 = \lambda\ w \to \exists[\ u \in \Sigma^*\ ]\ \exists[\ v \in \Sigma^*\ ](u \in L_1 \times v \in L_2 \times w \equiv u\ +\!\!+\ v)$.

Definition 7. Suppose L is a language, then we define L^n as the concatenation of L with itself over n times, i.e. $L^n = L \bullet L \bullet L ... L$. In Type Theory, it is defined as a recursive function where $L \wedge zero = [\![\epsilon]\!]$ and $L \wedge (suc\ n) = L \bullet (L \wedge n)$.

Definition 8. Suppose L is a language, then the closure of L, $L*$ is given by the set $\bigcup_{n \in N} L^n$. In Type Theory, we have $L \star = \lambda\ w \to \exists[\ n \in \mathbb{N}\ ]\ (w \in L \wedge n)$.

4.3 Regular Expressions and Regular Languages

The types of regular expressions and regular languages are defined in **RegularExpression.agda**.

Definition 9. Regular expressions over Σ are defined inductively 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 interpretation of regular expression in Agda is as follow:

```
data RegExp : Set where
  Ø   : RegExp
  ϵ   : RegExp
  σ   : Σ → RegExp
  _|_ : RegExp → RegExp → RegExp
  _·_ : RegExp → RegExp → RegExp
  _*  : RegExp → RegExp
```

The accepting language of regular expressions is defined as a function from *RegExp* to *Language*.

```
L^R : RegExp → Language
L^R 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

Recall that the set of all strings over Σ is defined as the type $List\ \Sigma^*$. However, this definition gives us no way to extract an ϵ alphabet from the input string. Therefore, we need to introduce another representation specific to this purpose.

Definition 10. We define Σ^e as the union of Σ and $\{\epsilon\}$, i.e. $\Sigma^e = \Sigma \cup \{\epsilon\}$.

The equivalent data type is as follow:

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

All the alphabets in Σ are included in Σ^e by using the α constructor while the ϵ alphabet corresponds to the constructor E in the data type.

Definition 11. 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 call these two lists as the ϵ-strings of the string 'Agda'. All the definitions, operations and lemmas regarding ϵ-strings can be found in **Language.agda**.

Now, let us define ϵ-NFA using Σ^{e*}. The type of ϵ-NFA is defined in **eNFA.agda** along with its operations and properties.

Definition 12. 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 $\mathcal{P}(Q)$ that 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.

It is formalised as a record in Agda as shown below:

```
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 is passed into the module as a parameter and Σ^e is constructed using *Sigma*. Together with Q, δ, q_0 and F, these five fields correspond to the 5-tuple ϵ-NFA. The other extra fields are used when computing ϵ-closures. They are $\forall qEq$ – a proof that any state in Q can reach itself by an ϵ-transition; $Q?$ – the decidable equality of Q; $|Q| - 1$ – the number of states minus 1; It – a vector of containing all the states in Q; $\forall q \in It$ – a proof that every state in Q is also in the vector It; and *unique* – a proof that there is no repeating elements in It.

Now, before we can define the accepting language of a given ϵ-NFA, we need to define several operations of ϵ-NFA.

Definition 13. A configuration is composed of a state and an alphabet from Σ^e, i.e. $C = Q \times \Sigma^e$.

Definition 14. A move in an ϵ-NFA N is represented by a binary function $(\vdash)$ on two configurations. We say that for all $w \in \Sigma^{e*}$ and $a \in \Sigma^e$, $(q, aw) \vdash (q', w)$ if and only if $q' \in \delta(q, a)$.

The binary function is defined in Agda as follow:

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

Definition 15. Suppose C and C' are configurations. We say that $C \vdash^0 C'$ if and only if $C = C'$; and $C_0 \vdash^k C_k$ 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 $0 \leq i < k$.

It is defined as a recursive function in Agda as follow:

```
_⊢ᵏ_−_ : (Q × Σᵉ*) → ℕ → (Q × Σᵉ*) → Set
(q , wᵉ) ⊢ᵏ zero − (q' , wᵉ')
  = q ≡ q' × wᵉ ≡ wᵉ'
(q , wᵉ) ⊢ᵏ suc n − (q' , wᵉ')
  = ∃[ p ∈ Q ] ∃[ aᵉ ∈ Σᵉ ] ∃[ uᵉ ∈ Σᵉ* ]
      (wᵉ ≡ aᵉ :: uᵉ × (q , aᵉ , uᵉ) ⊢ (p , uᵉ) × (p , uᵉ) ⊢ᵏ n − (q'
```

Definition 16. We say that $C \vdash^* C'$ if and only if there exists a number of chains n such that $C \vdash^n C'$.

Its corresponding type is defined as follow:

```
_⊢*_ : (Q × Σᵉ*) → (Q × Σᵉ*) → Set
(q , wᵉ) ⊢* (q' , wᵉ') = ∃[ n ∈ ℕ ] (q , wᵉ) ⊢ᵏ n − (q' , wᵉ')
```

Definition 17. For any string w, it is accepted by an ϵ-NFA if and only if there exists an ϵ-string of w that can take q_0 to an accepting state q. Therefore, the accepting language of an ϵ-NFA is given by the set $\{\ w \mid \exists w^e \in \Sigma^{e*}.\ w = to\Sigma^*(w^e) \wedge \exists q \in F.\ (q_0, w^e) \vdash^* (q, \epsilon)\ \}$.

The corresponding formalisation in Agda is as follow:

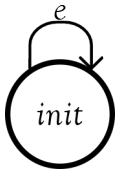
```
Lᵉᴺ : ϵ-NFA → Language
Lᵉᴺ nfa = λ w →
          ∃[ wᵉ ∈ Σᵉ* ] (w ≡ toΣ* wᵉ × (∃[ q ∈ Q ] (q ∈ᵈ F × (q₀ , w
```

4.5 Thompson's Construction

Now, let us look at the translation of regular expressions to ϵ-NFA which is defined in **Translation/RegExp-eNFA.agda**.

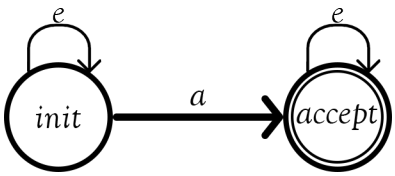
Definition 18. The translation of regular expressions to ϵ-NFA is defined inductively as follow:

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

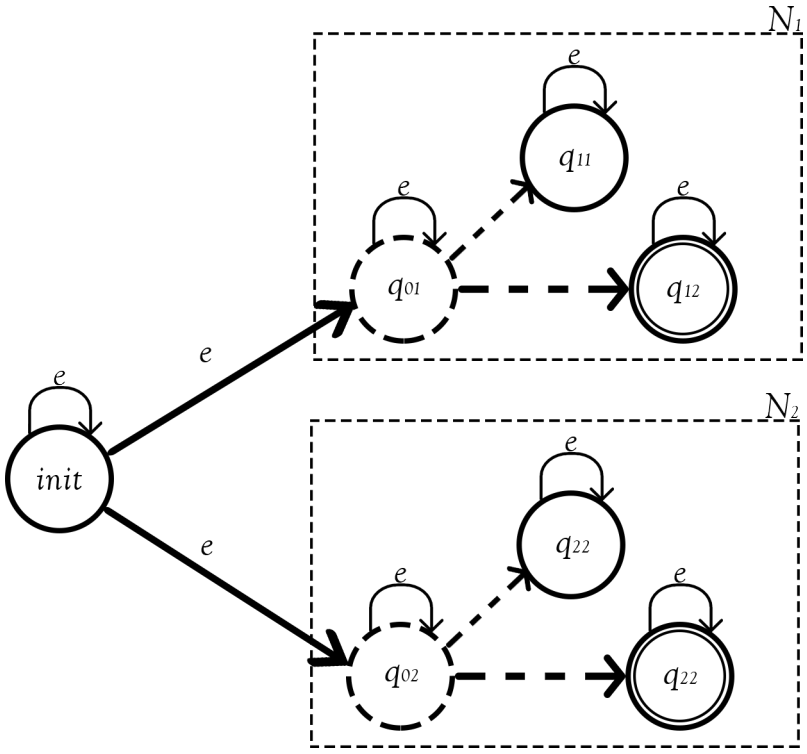


2. for ϵ, we have $M = (\{init\},\ \Sigma^e,\ \delta,\ init,\ \{init\})$ and graphically

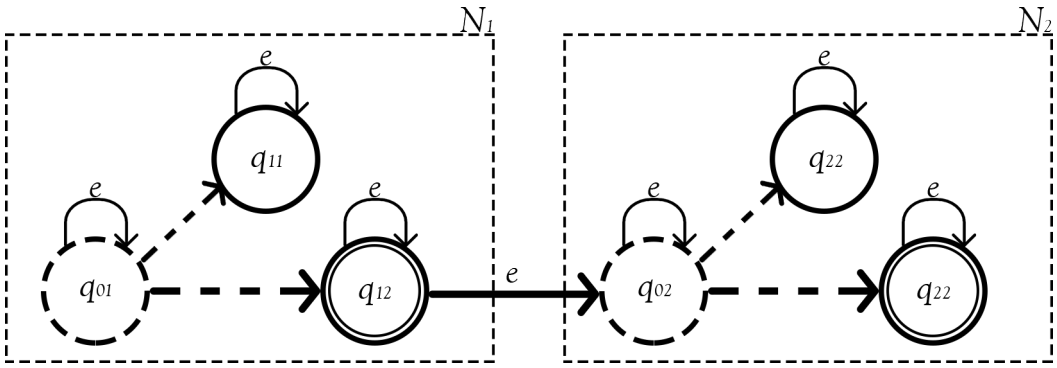
3. for a, we have $M = (\{init, accept\},\ \Sigma^e,\ \delta,\ init,\ \{accept\})$ and graphically



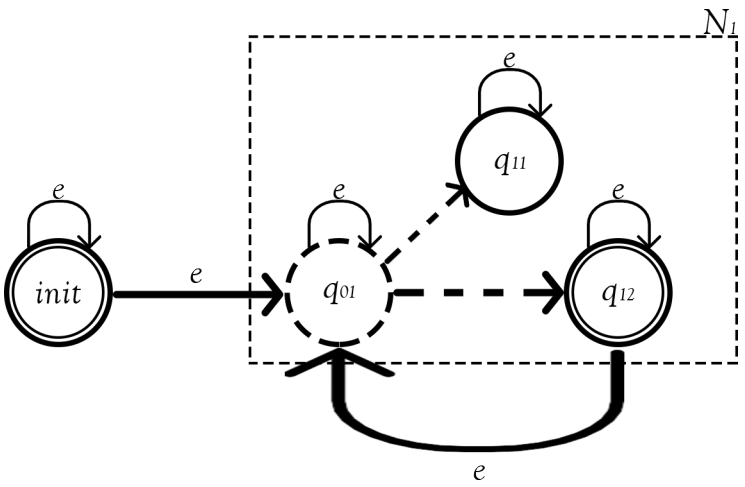
4. suppose $N_1 = (Q_1,\ \delta_1,\ q_{01},\ F_1)$ and $N_2 = (Q_2,\ \delta_2,\ q_{02},\ F_2)$ are ϵ-NFAs translated from the regular expressions e_1 and e_2 respectively, then
   (a) for $(e_1 \mid e_2)$, we have $M = (\{init\} \cup Q_1 \cup Q_2,\ \Sigma^e,\ \delta,\ init,\ F_1 \cup F_2)$ and graphically



(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



Now, let us prove the correctness of the above translation by proving their accepting languages are equal. The correctness proofs can be found in **Correctness/RegExp-eNFA.agda**.

Theorem 1. *For any given regular expression, e, its accepting language is equal to the language accepted by an ϵ-NFA translated from e using Thompson's Construction, i.e. $L(e) = L(\text{translated } \epsilon\text{-NFA})$.*

Proof. We can prove the theorem by induction on regular expressions.
Base cases. By Definition (18), it is obvious that the statement holds for $\emptyset$, ϵ and a.
Induction hypothesis 1. For any two 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 respectively. We assume that $L(e_1) = L(N_1)$ and $L(e_2) = L(N_2)$.
Inductive steps. There are three cases: 1) $e_1 \mid e_2$, 2) $e_1 \cdot e_2$ and 3) ${e_1}^*$.
1) *Case* $(e_1 \mid e_2)$: 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. Then for any string w,

1.1) if $(e_1 \mid e_2)$ accepts w, then by Definition (9) and Definition (5), 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 implies that there must exist an ϵ-string of w, w^e, which can take q_{01} to an accepting state q in N_1. Now, consider another ϵ-string of w, ϵw^e, it can take *init* to q in M because ϵ can take *init* to q_{01}. Recall that q is an accepting state in N_1; therefore, q is also an accepting state in M. Now, we have proved that there exists an ϵ-string w, ϵw^e, that can take *init* to an accepting state q in M; and thus M accepts w. The same argument also applies to the case when e_2 accepts w. Since we have proved that $w \in L(e_1 \mid e_2)$ implies $w \in L(M)$ for any string w; therefore $L(e_1 \mid e_2) \subseteq L(M)$ is true;

1.2) if M accepts w, then by Definition (17), there must exist an ϵ-string of w, w^e, which can take *init* to an accepting state q in M. q must be different from *init* because q is an accepting state but *init* is not. Now, by Definition (18), there are only two possible ways for *init* to reach q in M,

i) via q_{01} or ii) via q_{02}. Assuming case i), then the head of w^e must be an ϵ because it is the only alphabet that can take *init* to q_{01}. Furthermore, q is an accepting state in M; therefore, q is also an accepting state in N_1. Now, let $w^e = \epsilon u^e$, we have proved that there exists an ϵ-string w, u^e, which can take q_{01} to an accepting state q in N_1; and thus N_1 accepts w. By induction hypothesis, e_1 also accepts w; therefore, we have $w \in L(e_1 \mid e_2)$. The same argument also applies to case ii). Since we have proved that $w \in L(M)$ implies $w \in L(e_1 \mid e_2)$ for any string w; therefore $L(e_1 \mid e_2) \supseteq L(M)$ is true;

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

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

2.1) if $(e_1 \cdot e_2)$ accepts w, then by Definition (9) and Definition (6), there must exist a string $u \in L(e_1)$ and a string $v \in L(e_2)$ such that $w = uv$. By induction hypothesis, N_1 accepts u and N_2 accepts v. Therefore, there must exist i) an ϵ-string of u, u^e, which can take q_{01} to an accepting state q_1 in N_1 and ii) an ϵ-string of v, v^e, which can take q_{02} to an accepting state q_2 in N_2. Now, let us consider another ϵ-string of w, $u^e\epsilon\epsilon v^e$, it can take q_{01} to q_2 in M because u^e takes q_{01} to q_1, ϵ takes q_1 to mid, another ϵ takes mid to q_{02} and v^e takes q_{02} to q_2. Furthermore, q_2 is also an accepting state in M because q_2 is an accepting state in N_2. Therefore, we have proved that M accepts w. Since we have proved that $w \in L(e_1 \cdot e_2)$ implies $w \in L(M)$ for any string w; therefore $L(e_1 \cdot e_2) \subseteq L(M)$ is true;

2.2) if M accepts w, then by Definition (17), there must exist an ϵ-string of w, w^e, which can take q_{01} to an accepting state q in M. Since q is an accepting state in M; therefore, q must be in Q_2. The only possible way for q_{01} to reach q is by going through the state mid. This implies that there must exist i) an ϵ-string of u^e, which can take q_{01} to an accepting state q_1 in N_1 and ii) an ϵ-string v^e which can take q_{02} to q_2 in N_1 and $w^e = u^e v^e$. Let u and v be the normal strings of u^e and v^e respectively, then we have $u \in L(N_1)$, $v \in L(N_2)$ and $w = uv$. Now, by induction hypothesis, e_1 accepts u and e_2 accepts v; and thus $e_1 \cdot e_2$ accepts w. Since we have proved that $w \in L(M)$ implies $w \in L(e_1 \cdot e_2)$ for any string w; therefore $L(e_1 \cdot e_2) \supseteq L(M)$ is true;

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

3) *Case* ${e_1}^*$: Let $M = (Q,\ \delta,\ q_0,\ F) = (Q_1 \cup \{mid\} \cup Q_2,\ \delta,\ q_{01},\ F_2)$ be its translated ϵ-NFA. Then for any string w,

3.1) if $({e_1}^*)$ accepts w, then by Definition (9) and Definition (8), there must exist a number n such that $w \in L(e_1)^n$. Now, let us do induction on n.

Base case. When $n = 0$, then language L^0 can only accept the empty string ϵ; and thus $w = \epsilon$. From Definition (18), it is obvious that M accepts ϵ.

Induction hypothesis 2. Suppose there exists a number k such that $w \in L(e_1)^k$, then w is also accepted by M.

Induction steps. When $n = k + 1$, by Definition (6) and Definition (7), there must exist

a string $u \in L(e_1)$ and a string $v \in L(e_1)^k$ such that $w = uv$. By induction hypothesis (1), we have $u \in L(N_1)$ which implies that there must exist an ϵ-string u, u^e, that can take q_{01} to an accepting state q in N_1. Since q is an accepting state; an ϵ alphabet can take q back to q_{01}. Furthermore, by induction hypothesis (2), M also accepts v which implies that there exists an ϵ-string of v, v^e, that can take *init* to an accepting state p. Since the only alphabet that can take *init* to q_{01} is ϵ; therefore, v^e must be in the form of $\epsilon v'^e$. Now, we have proved that there exists an ϵ string of w, $\epsilon u^e \epsilon v'^e$, that can take *init* to an accepting state p in M; and thus M accepts w. Since we have proved that $w \in L({e_1}^*)$ implies $w \in L(M)$ for any string w; therefore $L({e_1}^*) \subseteq L(M)$ is true;

3.2) if M accepts w, then by Definition (17), there must exist an ϵ-string w, w^e, that can take *init* to an accepting state q using n transitions in M. If $init = q$, then w must be an empty string. By Definition (18), it is obvious that the empty string is accepted by ${e_1}^*$. If $init \neq q$, then there are only two possible ways for *init* to reach q: 1) from *init* to q without any transition that takes an accepting state in M back to q_{01}, we say that this path has no loops and 2) from *init* to q with at least one transition that takes an accepting state in M back to q_{01}, we say that this path has loop.

Case 1: Since $q \neq init$, then w^e must be in the form of $\epsilon w'^e$. Recall that the path has no loops, it is obvious that N_1 accepts w. Therefore by induction hypothesis (1), e_1 accepts w and thus ${e_1}^*$ also accepts w.

Case 2: Since $q \neq init$, then w^e must be in the form of $\epsilon w'^e$. Recall that the path has loops, we can separate w'^e into two parts: 1) an ϵ-string u^e that takes *init* to an accepting state p without loops and 2) an ϵ-string v^e that takes p to q_{01} to q with loops. Let u and v be the normal string of u^e and v^e respectively, then it is obvious that $w = uv$. By *case 1*, e_1 accepts u. Now, consider the path from q_{01} to q. The path must have less than n transitions; therefore, we can prove by induction the there must exist a number k such that $L(e_1)^k$ accepts v. Combining the above proofs, we have $w \in L({e_1}^*)$.

Since we have proved that $w \in L(M)$ implies $w \in L({e_1}^*)$ for any string w; therefore $w \in L({e_1}^*) \supseteq L(M)$ is true;

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

Therefore, by induction, $L(e) = L(\text{translated } \epsilon\text{-NFA})$ is true for all any regular expression e. □

4.6 Non-deterministic Finite Automata

Although the definition of NFA is very similar to that of ϵ-NFA, we will still give the definition of NFA separately. The type of NFA is defined in **NFA.agda** along with its operations and properties.

Definition 19. 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 $\mathcal{P}(Q)$ that 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.

It is formalised as a record in Agda as shown below:

```
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 is passed into the module as a parameter. Together with Q, δ, q_0 and F, these five fields correspond to the 5-tuple NFA. The other extra fields are used in powerset construction. They are $Q?$ – the decidable equality of Q; $|Q|-1$ – the number of states minus 1; It – a vector containing every state in Q; $\forall q \in It$ – a proof that every state in Q is also in the vector It; and *unique* – a proof that there is no repeating elements in It.

Now, before we can define the accepting language of a given NFA, we need to define several operations of NFA.

Definition 20. A configuration is composed of a state and an alphabet from Σ, i.e. $C = Q \times \Sigma$.

Definition 21. A move in an NFA N is represented by a binary function $(\vdash)$ on two configurations. We say that for all $w \in \Sigma^*$ and $a \in \Sigma$, $(q, aw) \vdash (q', w)$ if and only if $q' \in \delta(q, a)$.

The binary function is defined in Agda as follow:

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

Definition 22. Suppose C and C' are configurations. We say that $C \vdash^0 C'$ if and only if $C = C'$; and $C_0 \vdash^k C_k$ 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 $0 \leq i < k$.

It is defined as a recursive function in Agda as follow:

```
_⊢^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'
```

Definition 23. We say that $C \vdash^* C'$ if and only if there exists a number of chains n such that $C \vdash^n C'$.

Its corresponding type is defined as follow:

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

Definition 24. For any string w, it is accepted by an NFA if and only w can take q_0 to an accepting state q. Therefore, the accepting language of an NFA is given by the set $\{\ w \mid \exists q \in F.\ (q_0, w) \vdash^* (q, \epsilon)\ \}$.

The corresponding formalisation in Agda is as follow:

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

Now we can formulate the translation of ϵ-NFA to NFA by removing all the ϵ-transitions.

4.7 Removing ϵ-transitions

The translation of ϵ-NFA to NFA is defined in **Translation/eNFA-NFA.agda**. In order to remove all the ϵ-transitions, we need to know whether a state q can reach another state q' by only ϵ-transitions. Let us begin by defining such a predicate.

Definition 25. Let us first define a binary relation on states. We say that $q \to_\epsilon^0 q'$ if and only if q is equal to q'; and $q \to_\epsilon^k q'$ for $k \geq 1$ if and only if there exists a state p such that $p \in \delta(q, \epsilon)$ and $p \to_\epsilon^{k-1} q'$. We call this an ϵ-path from q to q'.

It is defined as a recursive function in Agda as follow:

```
_→ϵ^k_-_ : Q → ℕ → Q → Set
q →ϵ^k zero - q' = q ≡ q'
q →ϵ^k suc n - q' = ∃[ p ∈ Q ] ( p ∈^d δ q E × p →ϵ^k n - q' )
```

Definition 26. We say that $q \rightarrow_\epsilon^* q'$ if and only if there exists an ϵ-path from q to q' with n transitions, i.e. $\exists n.\ q \rightarrow_\epsilon^n q'$.

The corresponding type is as follow:

```
_→*ϵ_ : Q → Q → Set
q →*ϵ q' = ∃[ n ∈ ℕ ] q →ᵏϵ n − q'
```

Now we have to prove that for any two states q and q', $q \rightarrow_\epsilon^* q'$ is decidable. However, it is not possible to prove it directly because the set of natural numbers is infinite. Therefore, we will introduce an algorithm that computes the ϵ-closure for a state and prove that for any two states q and q', $q \rightarrow_\epsilon^* q'$ if and only if q' is in the ϵ-closure of q. By proving they are equivalent, we will have proved the decidability of $q \rightarrow_\epsilon^* q'$.

Definition 27. For any given state q, the ϵ-closure of q, i.e. $\epsilon\text{-}-closure(q)$ is obtained by:

1. put q into a subset S, i.e. $S = \{q\}$
2. loop for $|Q| - 1$ times:
   (a) for every state p in S, if ϵ can take p to another state r, i.e. $r \in \delta(p, \epsilon)$, then put r into S.
3. the result subset S is the ϵ-closure of q

Lemma 2. *For any two states q and q', $q \rightarrow_\epsilon^* q'$ if and only if $q' \in \epsilon\text{-}-closure(q)$.*

Proof. To prove the lemma, we have to prove that $q \rightarrow_\epsilon^* q'$ implies $q' \in \epsilon\text{-}closure(q)$ and $q' \in \epsilon\text{-}closure(q)$ implies $q \rightarrow_\epsilon^* q'$.

1) If $q \rightarrow_\epsilon^* q'$, then there must exist a number n such that ϵ^n can take q to q'. If $n < |Q|$, then it is obvious that $q' \in \epsilon\text{-}closure(q)$ is true. If $n \geq |Q|$, the ϵ-path from q to q' must have loop(s) inside. By removing the loop(s), the equivalent ϵ-path must have at most $|Q| - 1$ ϵ-transitions. Therefore, $q' \in \epsilon\text{-}closure(q)$ is true.

2) If $q' \in \epsilon\text{-}closure(q)$, it is obvious that $q \rightarrow_\epsilon^{|Q|-1} q'$ must be true. □

Since we have proved that they are equivalent, therefore the decidability of $q \rightarrow_\epsilon^* q'$ follows. Now, let us define the translation of ϵ-NFA to NFA using $q \rightarrow_\epsilon^* q'$.

Definition 28. For a given ϵ-NFA, $(Q,\ \delta,\ q_0,\ F)$, its translated NFA will be $(Q,\ \delta',\ q_0, F')$ where

- $\delta'(q, a) = \delta(q, a) \cup \{q' \mid \exists p \in Q.\ q \rightarrow_\epsilon^* p \wedge q' \in \delta(p, a)\}$; and
- $F' = F \cup \{q \mid \exists p \in Q.\ q \rightarrow_\epsilon^* p \wedge p \in F\}$.

Now, let us prove the correctness of the above translation by proving their accepting languages are equal. The correctness proofs can be found in Correctness/eNFA-NFA.agda.

Theorem 3. *For any ϵ-NFA, its accepting language is equal to the accepted language of its translated NFA, i.e. $L(\epsilon\text{-}NFA) = L(translated\ NFA)$.*

Proof. For a given ϵ-NFA, $\epsilon N = (Q,\ \delta,\ q_0,\ F)$, let its translated NFA be $N = (Q,\ \delta',\ q_0,\ F')$ according to Definition (28). To prove the theorem, we have to prove $L(\epsilon N) \subseteq L(N)$ and $L(\epsilon N) \supseteq L(N)$. For any string w,

1) if ϵN accepts w, then there must exist an ϵ-string of w, w^e, that can take q_0 to an accepting state q in n transitions. There are three possibilities: a) the last transition in the path is not an ϵ-transition, b) the path is divided into three parts, the first part from q_0 to a state p with less than n transitions; the second part from p to a state p_1 with an alphabet a and the third part from p_1 to q with only ϵ-transitions and c) the path from q_0 to q consists of only ϵ-transitions.

Case a: There must exist a state p and an ϵ-string, u^e, and an alphabet a such that u^e can take q_0 to p in less than n transitions, the alphabet a can take p to q and $w^e = u^e a$. We can prove by induction that the normal string of u^e, u, can take q_0 to p in N and $w = ua$. Therefore, we have proved that w can take q_0 to q in N and thus N accepts w.

Case b: There must exist two states p and p_1, an ϵ-string of w, u^e, and an alphabet a such that u^e can take q_0 to p in less than n transitions, a can take p to p_1, $p_1 \rightarrow_\epsilon^* q$ and $w = ua$. We can prove by induction that the normal string of u^e, u, can take q_0 to p in N. Furthermore, p_1 must be an accepting state in N because it can be transited to an accepting state q with only ϵ-transitions. Therefore, we have proved that w can take q_0 to an accepting state p_1 in N and thus N accepts w.

Case c: If the path from q_0 to q consists of only ϵ-transitions, then $q_0 \rightarrow_\epsilon^* q$; therefore, q_0 is also an accepting state. The accepted string must consist of ϵ only; therefore, $w = \epsilon$. It is obvious that N accepts ϵ.

2) if N accepts w, then w can take q_0 to an accepting state q. Since q is an accepting state, then q is also an accepting state in ϵN or there exist a state p such that $q \rightarrow_\epsilon^* p$ and $p \in F$. For the former case, since w is also an ϵ-string of itself; therefore, it is obvious that ϵN also accepts w. For the latter case, suppose the path from q to p has n ϵ-transitions, then we can append n $\epsilon's$ to w such that $w\epsilon^n$ can take q_0 to p in ϵN. Therefore ϵN accepts w. □

4.8 Deterministic Finite Automata

The type of DFA is defined in **DFA.agda** along with its operations and properties.

Definition 29. 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 that 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.

It is formalised as a record in Agda as shown below:

```
record NFA : Set₁ where
  field
    Q       : Set
    δ       : Q → Σ → DecSubset Q
    q₀      : Q
    F       : DecSubset Q
    _≈_     : Q → Q → Set
    ≈-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
```

The set of alphabets Σ is passed into the module as a parameter. Together with Q, δ, q_0 and F, these five fields correspond to the 5-tuple DFA. The other extra fields are used in proving its decidability. They are $\_ \approx \_$ – an equivalence relation on states; $\approx\text{-}isEquiv$ – a proof that the relation $\approx$ is an equivalence relation; $\delta\text{-}lem$ – a proof that for any alphabet a and any two states q and p, if $q \approx p$ then $\delta(q, a) \approx \delta(p, a)$; and $F\text{-}lem$ – a proof that for any two states q and p, if $q \approx p$ and q is an accepting state, then p is also an accepting state.

Now, before we can define the accepting language of a given DFA, we need to define several operations of DFA.

Definition 30. A configuration is composed of a state and an alphabet from Σ, i.e. $C = Q \times \Sigma$.

Definition 31. A move in an NFA N is represented by a binary function $(\vdash)$ on two configurations. We say that for all $w \in \Sigma^*$ and $a \in \Sigma$, $(q, aw) \vdash (q', w)$ if and only if $q' = \delta(q, a)$.

The binary function is defined in Agda as follow:

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

Definition 32. Suppose C and C' are configurations. We say that $C \vdash^0 C'$ if and only if $C = C'$; and $C_0 \vdash^k C_k$ 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 $0 \leq i < k$.

It is defined as a recursive function in Agda as follow:

```
_⊢^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 33. We say that $C \vdash^* C'$ if and only if there exists a number of chains n such that $C \vdash^n C'$.

Its corresponding type is defined as follow:

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

Definition 34. For any string w, it is accepted by an DFA if and only w can take q_0 to an accepting state q. Therefore, the accepting language of an DFA is given by the set $\{\ w \mid \exists q \in F.\ (q_0, w) \vdash^* (q, \epsilon)\ \}$.

The corresponding formalisation in Agda is as follow:

```
L^D : DFA → Language
L^D dfa = λ w →
            ∃[ q ∈ Q ] (q ∈^d F × (q₀ , w) ⊢* (q , [])))
```

Now we can formulate the translation of NFA to DFA by powerset construction.

4.9 Powerset Construction

The translation of NFA to DFA is defined in **Translation/NFA-DFA.agda**.

Definition 35. For any given NFA, $(Q,\ \delta,\ q_0,\ F)$, its translated DFA will be $(\mathcal{P}(Q),\ \delta',\ \{q_0\},\ F')$ where

- $\delta'(qs, a) = \{q' \mid \exists q \in Q.\ q \in qs \wedge q' \in \delta(q, a)\}$ and
- $F' = \{qs \mid \exists q \in Q.\ q \in qs \wedge q \in F\}$.

Now, before proving their accepting languages are equal, we first need to prove the following lemmas. Note that the theorems and proofs below can be found in **Correctness/NFA-DFA.agda**.

Lemma 4. *Let a NFA be $N = (Q,\ \delta,\ q_0,\ F)$ and its translated DFA be $D = (\mathcal{P}(Q),\ \delta',\ q_0, F')$ according to Definition (35). For any string w, if w can take q_0 to a state q with n transitions in N, then there must exist a subset qs such that $q \in qs$ and w can take $\{q_0\}$ to qs in D.*

Proof. We can prove the lemma by induction on n.
Base case. If $n = 0$, then $q_0 = q$ and $w = \epsilon$. It is obvious that the statement holds.
Induction hypothesis. For any string w, if w can take q_0 to a state q with k transitions in N, then there exists a subset qs such that $q \in qs$ and w can take $\{q_0\}$ to qs in D.
Induction step. If $n = k + 1$, then w can take q_0 to a state q by $k + 1$ transitions. Let $w = w'a$ where a is an alphabet, then w' can take q_0 to a state p by k transitions and a can take p to q. By induction hypothesis, there must exist a subset ps such that $p \in ps$ and w' can take $\{q_0\}$ to ps

in D. Furthermore, since a can take p to q, then a must be able to take ps to a subset qs where $q \in qs$. Therefore, there exists a subset qs such that $q \in qs$ and w can take $\{q_0\}$ to qs and thus the statement is true. □

Lemma 5. *Let a NFA be* $N = (Q,\ \delta,\ q_0,\ F)$ *and its translated DFA be* $D = (\mathcal{P}(Q),\ \delta',\ q_0, F')$ *according to Definition (35). For any string* w *and any two states* qs *and* ps *in* $\mathcal{P}(Q)$*, if* $(qs, w) \vdash^n (ps, \epsilon)$ *then* $ps = \{p \mid \exists q \in qs.\ (q, w) \vdash^n (p, \epsilon)\}$.

Proof. We can prove the lemma by induction on n.

Base case. If $n = 0$, then $qs = ps$ and $w = \epsilon$. Then for any state p in Q,

1) if $p \in ps$, then p also in qs. It is obvious that $(p, \epsilon) \vdash^0 (p, \epsilon)$ is true; therefore, $p \in \{p \mid \exists q \in qs.\ (q, w) \vdash^0 (p, \epsilon)\}$;

2) if $p \in \{p \mid \exists q \in qs.\ (q, w) \vdash^0 (p, \epsilon)\}$, then p must be in qs and thus $p \in ps$.

Induction hypothesis. For any subset qs and ps, any string w, If $(qs, w) \vdash^k (ps, \epsilon)$ then $ps = \{p \mid \exists q \in qs.\ (q, w) \vdash^k (p, \epsilon)\}$.

Induction step. If $n = k + 1$, then w must be able to take qs to ps with $k + 1$ transitions in D. Therefore there must exist a subset rs such that an alphabet a can take qs to rs, i.e. $rs = \delta'(qs, a)$, and a string u can take rs to ps with k transitions. By induction hypothesis, we have $ps = \{p \mid \exists r \in rs.\ (r, u) \vdash^k (p, \epsilon)\}$. Then for any state p in Q,

1) if $p \in ps$, there exists a state $r \in rs$ such that $(r, u) \vdash^k (p, \epsilon)$. Since $rs = \delta'(qs, a)$; therefore, $r \in \delta'(qs, a)$ and thus there exists a state $q \in qs$ such that $r \in \delta(q, a)$. Therefore, $(q, w) \vdash^{k+1} (p, \epsilon)$ is true and thus $p \in \{p \mid \exists q \in qs.\ (q, w) \vdash^{k+1} (p, \epsilon)\}$.

2) if $p \in \{p \mid \exists q \in qs.\ (q, w) \vdash^{k+1} (p, \epsilon)\}$, then there exists a state $q \in qs$ such that $(q, w) \vdash^{k+1} (p, \epsilon)$. Also, there must exist a state $r \in Q$ such that $r \in \delta(q, a)$ and the string u can take r to p. Since, $q \in qs$ and $r \in \delta(q, a)$; therefore, $r \in \delta'(qs, a) = rs$ and thus $p \in \{p \mid \exists r \in rs.\ (q, w) \vdash^k (r, \epsilon)\} = ps$. □

Now, by using the above lemmas, we can prove the correctness of the translation by proving their accepting languages are equal.

Theorem 6. *For any NFA, its accepting language is equal to the language accepted by its translated DFA, i.e.* $L(NFA) = L(\textit{translated DFA})$.

Proof. For a given NFA, $N = (Q,\ \delta,\ q_0,\ F)$, let its translated DFA be $D = (\mathcal{P}(Q),\ \delta',\ q_0,\ F')$. To prove the theorem, we have to prove that $L(N) \subseteq L(D)$ and $L(N) \supseteq L(D)$. For any string w,

1) if N accepts w, then w can take q_0 to an accepting state q with n transitions in N. By Theorem (4), there must exist a subset qs such that $q \in qs$ and w can take $\{q_0\}$ to qs in D. Since q is an accepting state; therefore, qs is also an accepting state in D and thus D accepts w.

2) if D accepts w, then w can take $\{q_0\}$ to an accepting state qs in D with n transitions. Since qs is an accepting state, therefore, there must exist a state q in Q such that $q \in qs$ and q is also an

accepting state in N. Assuming w cannot take q_0 to q in N with n transitions, then by Theorem (5), $q \notin qs$ which contradicts the fact that $q \in qs$. Therefore, w must be able to take q_0 to q in N with n transitions and thus N accepts w. □

4.10 Decidability of DFA and Regular Expressions

The decidability of DFA cannot be proved directly using the current representation because $\mathbb{N}$ is infinite. Therefore, we have to introduce another representation for its accepting language and prove that it is equivalent to the original representation. The representation and the proofs can be found in **DFA.agda**.

Definition 36. We define a function δ^* that takes a state q and a string w as the arguments and returns a state p after running the DFA. It is defined recursively as follow:

```
δ* : Q → Σ* → Q
δ* q [] = q
δ* (a :: w) = δ* (δ q a) w
```

Definition 37. We define δ_0 as the function that runs the DFA from q_0 with a string w.

```
δ₀ : Σ* → Q
δ₀ w = δ* q₀ w
```

Now, we have to prove that it is equivalent to the original definition.

Lemma 7. *For any string w, $\delta_0(w) \in F$ if and only if $\exists q \in Q.\ q \in F \wedge (q_0, w) \vdash^* (q, \epsilon)$*

Proof. ... □

Now, we can prove that the accepting language of a given DFA is decidable by using the above definitions and lemmas.

Theorem 8. *For any DFA, its accepting language is decidable, i.e. $\forall w.\ w \in L(DFA)$ is decidable.*

Proof. ... □

Since we have also proved that the accepting language of regular expressions and DFA are equal; therefore, the accepting lanugage of regular expression must also be decidable.

Theorem 9. *For any given regular expression, e, its accepting language is decidable, i.e. $\forall w.\ w \in L(e)$ is decidable.*

Proof. ... □

4.11 Minimal DFA

The definition of minimal DFA can be found in **Correctness/DFA-MDFA.agda**.

In order for a DFA to be minimal, it must satisfies two criteria: 1) every state must be reachable from the start state and 2) the states cannot be further reduced. Criteria 1) is straight-forward, here is the definition.

Definition 38. For any state q, if there exists a string w that can take q_0 to q, then q is reachable.

For criteria 2), we have to introduce a binary relation of states.

Definition 39. For any two states p and q, we say that a string w can distinguish p and q if and only if exactly one of $\delta^*(p, w)$ and $\delta^*(q, w)$ is an accepting state.

Definition 40. For any two states p and q, we say that $p \nsim q$ if and only if there exists a string w that can distinguish p and q.

Now, we can define the irreducibility of a given DFA.

Definition 41. For a given DFA, it is irreducible if and only if for any two states p and q, if p is not equal to q, then $p \nsim q$.

We can now define minimal DFA.

Definition 42. For a given DFA, it is minimal if and only if all its states are reachable and it is irreducible.

Now, let us formulate the translation of DFA to MDFA.

4.12 Minimising DFA

There are two procedures in minimising a DFA: 1) removing all the unreachable states to construct a RDFA and 2) perfore quotient construction on the RDFA to build a MDFA.

4.12.1 Removing unreachable states

...

4.12.2 Quotient construction

...

5 Further Extensions

In this section, we will discuss two possible extentions to our project: 1) Myhill-Nerode Theorem and 2) Pumping Lemma. In these two theorems, they both use the translation of regular expressions to DFA in thier arguments. Therefore, they can be built on top of our formalisation.

Myhill-Nerode Theorem In general, Myhill-Nerode Theorem states that a language L is regular if and only if ...

Pumping Lemma (for regular languages) In general, the Pumping Lemma states that ...

6 Evaluation

Throughout the project, we have made several decisions over the representations of mathematical objects such as subsets. We will disucss their consequences in this section. Furthermore, we will also review the whole development process. At the end of this section, we will also discuss the feasibility of formalising mathematics and programming logic in practice.

6.1 Different choices of representations

In computer proofs, an abstract mathematical object usually requires a concrete representation. 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 representations we have chosen and their consequences.

The set of states (Q) and its subsets As we have mentioned in section 3, Firsov and Uustalu [8] represent the set of states (Q) and its subsets as column vectors. However, this representation looks unnatural compare to the actual mathematical definition. Therefore, at the beginning, we intended to avoid the vector representation and to represent the sets in abstract forms. In our approach, the set of states is represented as a data type in Agda, i.e. $Q : Set$, and its subsets are represented as unary functions on Q, i.e. $DecSubset\ Q = Q \to Bool$.

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 [8]. However, problems arise when the sets have to be iterated when computing the ϵ-closures because it is impossible to iterate the sets using this representation. In order to solve the problems, several extra fields are added into the definition of automata such as It – a vector containing all the states in Q. With It, a subset of Q can be iterated by applying it own function, $Q \to Bool$, to all the elements in It. Note that the vector It is equivalent to the vector representation of Q.

The accepting languages of regular expressions and finite automata At first, the accepting language of regular expression was defined as a decidable subset, i.e. $L^R : RegExp \to DecSubset\ \Sigma^*$. The decision was reasonable because the language has been proved decidable for many years. However, L^R is also a boolean decider for regular expressions and this definition added a great amount of difficulties in writing the proofs because the proofs had to be built on top of the decider. For example, in the concatenation case, an extra recursive function was needed to iterate different combinations of input string and correctness proof of this function is difficult to complete. However, the decidability of the languages is actually not necessary in proving their equality. Therefore, in the current version of our Agda code, the language is defined using the

general subset, i.e. $L^R : RegExp \to Subset\ \Sigma^*$. This definition allows to prove the equality of the languages in an abstract level.

The same situation also applies 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 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. Therefore, the accepting language of NFA is now defined in an abstract form.

However, this does not mean that we can ignore all the algorithms. In fact, we still need to design algorithm in computing ϵ-closures and powerset construction. Here is our approach: 1) separate the algorithm from the mathematical definition and 2) prove that they are equivalent. In this way, the decidability of the mathematical definition will follow from that of the algorithm. The advantage is that it is easier to use the abstract mathematical definition in other proofs than using the algorithm.

The set of reachable states from q_0 Let us recall the definition of reachable states from 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 q is reachable if and only if there exists a string w that can take q_0 to q. Therefore, the set Q^R should contains all and only the reachable states in Q. However, there may exist more than one reachability proof for the same state. Therefore, for a state in Q, there may be more than one element in Q^R and thus Q^R may be larger than the original set Q or even worse, it may be infinite. This leads to a problem when a DFA is constructed using Q^R as the set of states. If Q^R is infinite, then it is impossible to iterate the set during quotient construction. Even if the set 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 to the proof of $L(\text{DFA}) = L(\text{MDFA})$ because we can provide an equality relation of states in the DFA. In the translation from DFA to MDFA, we defined the equality relation as follow: any two elements in Q^R are equal if and only if their 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 (w) sorted in alphabetical order that can take q_0 to q. Another solution is to use Homotopy Type to declare the set Q^R. This type allows us to group different reachability proofs into a single element such that every state will only appear once in Q^R.

6.2 Development Process

As we have mentioned before, the parts related to quotient construction were not completed. 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 when converting regular expressions to NFA 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 to write 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 be 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 development and the time should have been spent more wisely on other parts of the project such as powerset construction or the report.

6.3 Computer-aided verification in practice

In order to evaluate the feasibility 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 [9], 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 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 work properly, then any formalised proof that can be compiled without errors is said to be correct. Furthermore, a proof can been seen as a group 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 w^e can take q_0 to another state $inj\ q_1$ in an ϵ-NFA translated from a regular expression e^*, then there exists a number n_1 and an ϵ-string u^e that will take $inj\ q_{01}$ to $inj\ q_1$ where q_{01} is the start state of e and q_1 is a state in e. There are several techniques used in the proof; for example, 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. ... [more]

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 as there are many similarities between writing codes and writing proofs; for example, pattern matching compare to case analysis and recursive function compare to

mathematical induction. Furthermore, Agda is convenient to use for its interactive features. The features allow us to 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 may be very difficult to implement under this limitation. For example, the usual algorithm that is used to compute the ϵ-closure for a state q is as follow:

1. put q into a subset S, i.e. $S = \{q\}$
2. for every state p in S, if another state r is reachable from p with an ϵ, i.e. $r \in \delta(p,\ \epsilon)$, then put r into S.
3. loop (2) until no new elements is added to S
4. the resulting subset S is the ϵ-closure of q

This kind of algorithms cannot be implemented directly without modifications. ... [more]

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 [14] has raised the notion of *Proof Carrying code*. The idea is to accompany program with several proofs that proves the program specifications within the same platform. In fact, there is already an extraction mechanism [12] in Coq that allows us to extract proofs and functions in Coq into Ocaml, Haskell or Scheme programs.

... [programming logic in concurrency, distributive systems, cloud]

7 Conclusion

...

Bibliography

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

[2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

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

[4] Ana Bove and Peter Dybjer. Dependent types at work. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, pages 57–99. Springer-Verlag, Berlin, Heidelberg, 2009.

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

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

[7] Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. A constructive theory of regular languages in coq. In *Proceedings of the Third International Conference on Certified Programs and Proofs - Volume 8307*, pages 82–97, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[8] Denis Firsov and Tarmo Uustalu. Certified parsing of regular languages. In *Proceedings of the Third International Conference on Certified Programs and Proofs - Volume 8307*, pages 98–113, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

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

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

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

[12] 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.

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

[14] 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.

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

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

[17] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[19] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

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

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

Appendices

... [zip file desc]