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]. ... [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 on 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 [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 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.

... [equivalent haskell boolean data type for comparison]

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 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 (*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
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 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, a function in Agda must pattern match on all possible cases. 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 kind 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
```

[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.

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

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. Different natural numbers will give different types produced by 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 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 $(Vec\ A\ (suc\ n))$ 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. 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, 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 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.

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. A predicate 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
    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 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 (step n nIsEven)
... | inj₂ nIsOdd = inj₁ (step 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 $\exists[x \in A]\ B$. As an example, we can 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 equality 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 $\exists n \in \mathbb{N}.\ n = 1 + 1$ as follow:

```
lem₃ : ∃[ n ∈ ℕ ] n ≡ (1 + 1)
lem₃ = suc (suc zero) , 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 use an 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, we only consider the list of natural numbers. 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

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 a partial automaton. It was implemented using the *Maybe* type as follow:

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

When a string is accepted by the regular expression, i.e. $w \in L(e)$, the decider will return its proof. However it fails to generate a proof for the opposite case, i.e. $w \notin L(e)$. 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 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 can generate proofs for both cases. In their definition of NFA, the set of states Q and its subsets are represented as vectors while 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 $\_ * \_$ is an inductive family that takes two natural numbers n and m and produces a matrix type $n \times m$. 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 our formalisation: 1) the translation of regular expressions to DFA and 2) the correctness proofs of the translation. The translation in 1) is divided into several steps. Firstly, any regular expression is converted into an ϵ-NFA using Thompson's construction [18]. Secondly, all the ϵ-transitions are removed by computing the ϵ-closures in order to get a NFA. 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. In part 2), we prove that accepting languages of the translated output are equal. i.e. $L(regex) = L(translated\ \epsilon\text{-NFA}) = L(translated\ \text{DFA}) = L(translated\ \text{MDFA})$. Furthermore, we also need to prove that the translated MDFA is minimal.

In this section, 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 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

Subset and decidable subset are defined in **Subset.agda** and **Subset/DecidableSubset.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.

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 $sub = \lambda\ (x : A) \to ?$. The ? here should be replaced by the conditions for x to be included in sub. This construction is very similar to set comprehension. For example, the subset $\{x \mid x \in A,\ P(x)\}$ corresponds to $\lambda\ (x : A) \to P\ x$. Furthermore, sub is also a predicate on A as it has the type $A \to Set$. As we have mentioned before, its decidability will remain unknown until it is 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 definfed 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.

4.1.1 Operations on Subsets

...

4.1.2 Operations on Decidable Subsets

...

4.2 Languages

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

Suppose Σ is a set of alphabets, then it can be represented as a data type in Type Theory, i.e. $\Sigma : Set$. Note that the equality of the set Σ needs to be decidable. In Agda, they are passed to every module as parameters in the form $(\Sigma : Set)$ $(dec : DecEq\ \Sigma)$.

Definition 3. We first define Σ^* as the set of all strings over Σ. In our approach, it is expressed as a list, 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 of the input string can be extracted 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 defined as the set $\{w \mid w \in L_1 \ \vee \ w \in L_2\}$. In Type Theory, it is defined as $L_1 \cup L_2 = \lambda\ w \rightarrow w \in L_1 \ \uplus \ w \in L_2$.

Definition 6. Suppose L_1 and L_2 are languages, then the concatenation of the two languages, $L_1 \bullet L_2$, is defined as $\{w \mid \exists u \in L_1.\ \exists v \in L_2.\ w = uv\}$. In Type Theory, it is defined as $L_1 \bullet L_2 = \lambda\ w \rightarrow \exists[\ u \in \Sigma^*\ ]\ \exists[\ v \in \Sigma^*\ ](u \in L_1 \times v \in L_2 \times w \equiv u \mathbin{++} v)$.

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

Definition 8. Suppose L is a language, then the closure of L, $L*$ is defined as $\bigcup_{n \in N} L^n$. In Type Theory, it is defined as $L \star = \lambda\ w \rightarrow \exists[\ n \in \mathbb{N}\ ](w \in L\ \hat{}\ n)$.

4.3 Regular Expressions and Regular Languages

Regular expression and regular language are defined in **RegularExpression.agda**. The set of alphabet Σ is passed to the file as a parameter.

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 is intuitive:

```
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 strings is defined as *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 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 Σᵉ : Set where
  α : Σ → Σᵉ
  E : Σᵉ
```

Alphabets in Σ are included in Σ^e by using the α constructor while the ϵ alphabet correpsonds to 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'. The definitions, operations and lemmas regarding ϵ-strings can be found in **Language.agda**.

Now, let us define ϵ-NFA using Σ^{e*}. 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.

ϵ-NFA is formalised in Agda as a record.

```
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 $\Sigma : Set$ is passed into the module as a parameter. 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 of the lemma that says 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 length $|Q|$ that contains every state in Q; $\forall q \in It$ – a proof of the lemma that says every state in Q is also in the vector It and *unique* – a proof of the lemma that says there is no repeating elements in It. We will look into these extra fields with more details in later part.

In order to define the accepting language of a given ϵ-NFA, we need to define several operations of ϵ-NFA first.

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 $(q, aw) \vdash (q', w)$ for all $w \in \Sigma^{e*}$ and $a \in \Sigma^e$ 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 $C = C'$. Also, $C_0 \vdash^k C_k$ 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 $0 \leq i \leq k$.

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' , wᵉ'))
```

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 N if and only if there exists an ϵ-string w^e of w that can take q_0 to an accepting state q.

Definition 18. The language accepted by an ϵ-NFA is given by the set $\{\ w \mid \exists w^e \in \Sigma^{e*}.\ w = to\Sigma^*(w^e) \wedge \exists q \in F.\ (q_0\ ,\ w^e) \vdash^* (q\ ,\ \epsilon)\ \}$.

The correpsonding formalisation in Agda is defined as follow:

```
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 , []))))
```

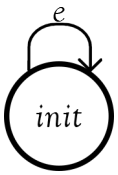
Now we can formulate the translation of regular expressions to ϵ-NFA using Thompson's Construction and prove that their acceping languages are equal.

4.5 Thompson's Construction

The translation of regular expressions to ϵ-NFA is defined in **Translation/RegExp-eNFA.agda**.

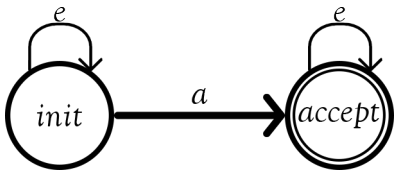
Definition 19. The translation of regular expressions to ϵ-NFA is defined inductively as follow:

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

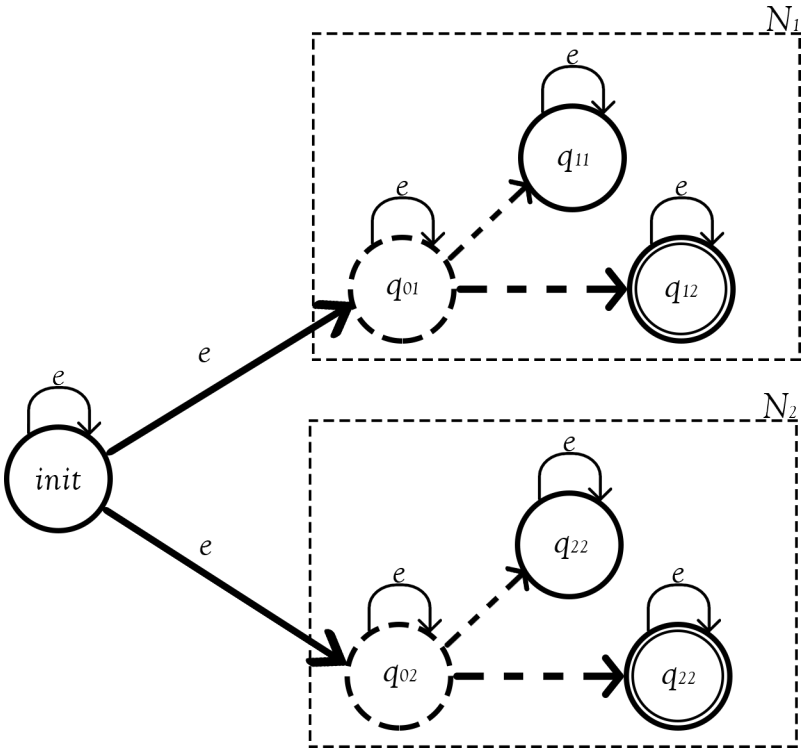


2. for ϵ, we have $M = (\{init\},\ \Sigma^e,\ \delta,\ init,\ \{init\})$ and graphically

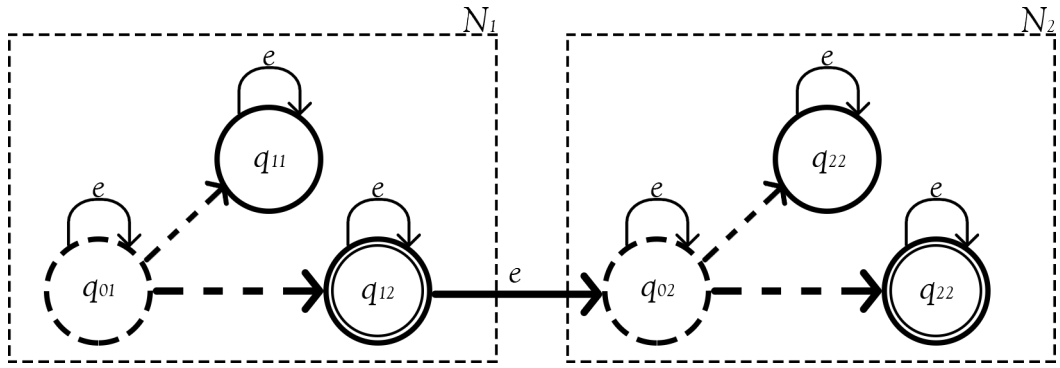
3. for a, we have $M = (\{init, accept\},\ \Sigma^e,\ \delta,\ init,\ \{accept\})$ and graphically



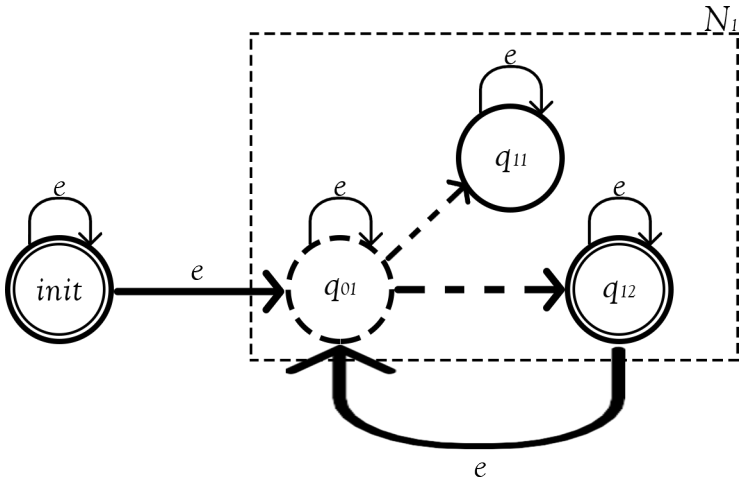
4. if $N_1 = (Q_1,\ \delta_1,\ q_{01},\ F_1)$ and $N_2 = (Q_2,\ \delta_2,\ q_{02},\ F_2)$ are ϵ-NFAs translated from the regular expressions e_1 and e_2 respectively, then
   (a) for $(e_1 \mid e_2)$, we have $M = (\{init\} \cup Q_1 \cup Q_2,\ \Sigma^e,\ \delta,\ init,\ F_1 \cup F_2)$ and graphically



(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



Now, let us prove the correctness of the above translation by proving their accepting languages are equal. The correctness proofs are defined 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. $L(e) = L(translted\ \epsilon\text{-}NFA)$.*

Proof. In order to prove the above theorem, we need to prove that for any regular expression e, $L(e) \subseteq L(translated\ \epsilon\text{-}NFA)$ and $L(e) \supseteq L(translated\ \epsilon\text{-}NFA)$. They can be proved by induction on regular expressions.

Base cases. By Definition ?, 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 can 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 ?, 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 w^e of w which can take q_{01} to an accepting state q in N_1. Now, we construct another ϵ-string of w by adding an ϵ before the ϵ-string w^e, i.e. ϵw^e. The ϵ-string ϵw^e 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 ϵw^e of w that can take *init* to an accepting state q in M; and thus M accepts w. The same argument also applies to other case when e_2 accepts w. Since we have proved that $w \in L(e_1 \mid e_2)$ imples $w \in L(M)$ for any string w; therefore $L(e_1 \mid e_2) \subseteq L(M)$ is true;

1.2) if M accepts w, then there must exist an ϵ-string w^e of w 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 ?, 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 u^e of w which can take q_{01} to an accepting state q in N_1; and thus N_1 accpets w. By inductoin 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 ?, 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 u^e of u which can take q_{01} to an accepting state q_1 in N_1 and ii) an ϵ-string v^e of v which can take q_{02} to an accepting state q_2 in N_2. Now, let us consider an ϵ-string $u^e\epsilon\epsilon v^e$ of w, $u^e\epsilon\epsilon v^e$ 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 $w \in L(e_1 \cdot e_2)$ imples $w \in L(M)$ 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 ?, there must exist an ϵ-string w^e of w 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 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. 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 $w \in L(M)$ imples $w \in L(e_1 \cdot e_2)$ 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 there must exist a number n such that $w \in (L\ \hat{}\ n)$. Now, lets do induction on n.

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

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

Induction steps. When $n = k + 1$, by Definition ?, there must exist a string $u \in L(e_1)$ and a string $v \in L(e_1)\ \hat{}\ k$ 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^e of u 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 v^e of v 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 $\epsilon u^e \epsilon v^{e'}$ 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 $w \in L(e_1\ ^*) \subseteq L(M)$ is true;

3.2) if M accepts w, then there must exist an ϵ-string w^e of w that can take $init$ to an accepting state q in M. If $init = q$, then w must be an empty string. By Definition ?, 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 the ϵ-string into two parts: 1) a string u^e that takes $init$ to an accepting state p without loops and 2) a 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 2*, e_1 accepts u. Also, by induction, we can prove that there exists a number n such that $(e_1)\ \hat{}\ n)$ accepts v. Combining the aboves, 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, $L(e) = L(translted\ \epsilon\text{-NFA})$ is true forall 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 define NFA separately for the sake of integrity. NFA is defiend in **NFA.agda** along with its operations and properties.

Definition 20. 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.

It is formalised in Agda as a record.

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

The set of alphabets $\Sigma : Set$ is passed into the module as a parameter. 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 minus 1. 'It' is a vector of length $|Q|$ containing all the states in Q. $\forall q \in It$ is a proof of the lemma – all the states in Q are also in the vector 'It'. *unique* is a proof of the lemma – there is no repeating elements in 'It'. These extra fields are important when performing powerset construction, we will look into them again in later part.

Now, we want to define the set of strings Σ^* accepted by a given NFA, but before that, we will first define several operations of NFA.

Definition 21. A configuration is composed of a state and an alphabet from Σ, i.e. $C = Q \times \Sigma$.

Definition 22. A move in an NFA N is represented by a binary function $\vdash$ on configurations. We say that $(q, aw) \vdash (q', w)$ for all $w \in \Sigma$ and $a \in \Sigma$ if and only if $q' \in \delta(q, a)$.

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

Definition 23. Suppose C and C' are configurations. We say that $C \vdash^0 C'$ if and only if $C = C'$. Also, $C_0 \vdash^k C_k$ 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^e ≡ w^e'
(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 24. 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 25. For any string w, it is accepted by an NFA N if and only if w can take q_0 to an accepting state q.

Definition 26. The language accepted by an NFA is given by the set $\{\ w \mid \exists q \in F.\ (q_0\ ,\ w) \vdash^* (q\ ,\ \epsilon)\ \}$.

```
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 **eNFA-NFA.agda**. In order to remove all the ϵ-transitions, we need to know whether a state q can reach another state q' with only ϵ transitions.

Definition 27. We say that $q \to_\epsilon^0 q'$ if and only if q is equal to q'. Also, $q \to_\epsilon^k q'$ for $k > 1$ if and only if there exists a state p such that $p \in \delta(q,\ \epsilon)$ and $p \to_\epsilon^{k-1} q'$.

```
_→ϵ^k_−_ : Q → ℕ → Q
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 28. We say that $q \to_\epsilon^* q'$ if and only if there exists a number n such that $q \to_\epsilon^n q'$.

```
_→ϵ*_ : Q → Q
q →ϵ* q' = ∃[ n ∈ ℕ ] q →ϵ^k n − q'
```

Now we have to prove that for any pair of states $(q,\ q')$, $q \to_\epsilon^* q'$ is decidable.

4.8 Deterministic Finite Automata

...

4.9 Powerset Construction

...

4.10 Minimal DFA

...

4.11 Minimising DFA

...

4.11.1 Removing unreachable states

...

4.12 Quotient construction

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 represent the set of states Q and its subsets as column matrices in their work [7]. 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 are 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 [7]. However, there is a problem when the sets have to be iterated when computing the ϵ-closures because this representation is not possible to iterate. Therefore, in the current version, several extra fields are included in the automata including It – a vector containing all the states in Q. With It, the subsets of Q can be iterated by applying their own functions $Q \to Bool$ to all the elements in It. Note that the vector 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. $L^R : RegExp \to DecSubset\ \Sigma^*$. 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. The definition 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. $L^R : RegExp \to Subset\ \Sigma^*$. 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 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 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 q_0.

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

data Q^R : Set where
  reach : ∀ q → Reachable q → Q^R
```

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 a single state. This implies that there may be more than one element in Q^R having the same state in Q. Therefore, 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 there is no possible way 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 on the proof of $L(DFA) = L(MDFA)$ 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 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 (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 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. 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 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 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 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 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 as there are many similarities between writing codes and writing proofs, for examples, 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 know what is happening inside the proof body easily 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 ϵ−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 ϵ−transition, we put it in A
> 4) The resulting set A is the ϵ−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.

programming logic like concurrency? what about distributive system? cloud? ...

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 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] 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.

[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. 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.

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

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

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

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

Appendices

intro to the files