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 [1] in Type Theory with the aid of a dependently-typed functional programming language, Agda [13]. Automata Theory is an extensive work; therefore, it will be unrealistic to include all the materials under the time constraint. Accordingly, this project will only focus on the theorems and proofs that are related to the translation between regular expressions and finite automata. In addition, this project also gives a brief introduction on how complex and non-trival proofs are formalised.

Our Agda formalisation is consist of two major 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 programming as well as in formalising programming logic in Agda. ...

1.2 Overview

In the beginning of section two, we will give a brief introduction on Agda and dependent types. In addition, we will also look into some small Agda proofs so that readers can have a taste of how proofs are formalised in Type Theory. In the end of section two, we will describe a similar reseach conducted by Firsov and Uustalu [6].

Following the background, the third section will be a detail description of our work. In this section, we will walk through the Agda formalisation of the two major components. All the definitions, theorems and proofs written in this report are extracted from our Agda code to adapt the environment in Type Theory.

In the fourth section, we will discuss two possible extensions to the project: 1) Myhill-Nerode Theorem and 2) the Pumping Lemma. After that, we will evaluate the project as a whole. Finally, the conclusions will be drawn.

2 Background

2.1 Agda

Agda is a dependently-typed functional programming language and a proof assistant based on Intuitionistic Type Theory [9]. The current version (Agda 2) is rewritten by Norell [11] during his 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 [3] and [12]. Interested readers can take a look in order to get a more precise idea on how to work with Agda. Now, let us begin by showing how to do ordinary functional programming in Agda.

2.1.1 Simply Typed Functional Programming

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

Boolean We first define the type of boolean values in Agda.

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

Bool is a data type which has two constructors: *true* and $false$. Note that these two constructors are also elements of *Bool* as they take no arguments. On the other hand, *Bool* 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 function over boolean values.

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

In Agda, we must explicitly provide a type declaration for every variable. Note that we can declare partial functions in Haskell but not in Agda. For instance, the function below will be rejected by the Agda compiler.

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

Natural Number Now, let us define the type of natural numbers inductively as follow:

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

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

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

Parameterised Types In Haskell, the type of list $[a]$ is parameterised by the type parameter a. The analogus 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 the function *head* which will return the first element of a given list.

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

In Haskell, we can simply skip the case [ ] and it will produce an error for us. However, as we mentioned before, we have to pattern match every case in Agda. One possible workaround is to use the *Maybe* type. However, we can do a better job with dependent types.

2.1.2 Dependent Types

Dependent types are types that depend on values of other types. For example, A^n is a vector that contains n elements of A. This type is not possible to be declared in simply-typed systems like Haskell or Ocaml. Now, let us look at how it is defined in Agda using dependent type.

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

In the type declaration $data\ Vec\ (A : Set) :\ \mathbb{N} \to Set$, $A : Set$ is the type parameter defining the type of elements that will be in the vector. While the part $\mathbb{N} \to Set$ means that Vec takes a number n from the set $\mathbb{N}$ and produces a type that depends on n. With different natural number n, we get different type from the inductive family Vec. For example, $Vec\ A\ zero$ is the type of empty vectors and $Vec\ A\ 10$ is another vector type with length ten.

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

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

We only need to pattern match on the case $(x :: xs)$ because the type $Vec\ A\ (suc\ n)$ ensures that the argument will never be [ ]. Apart from vectors, we can also define the type of binary search tree in which every tree is guranteed to be sorted. However, we will not be looking into that in here as it is not our major concern. Interersted readers can take a look at Section 6 in [3]. Furthermore, dependent types also allow us to encode predicate logic and program specifications as types. Before we go into these two applications, we will first introduce the idea of propositions as types.

2.1.3 Propositions as Types

In the 1930s, Curry identified the correspondence between propositions in propositional logic and types [4]. After that, in the 1960s, de Bruijn and Howard extended Curry's correspondence to predicate logic by introducing dependent types [5,8]. Later on, Martin-Löf published his work, Intuitionistic Type Theory [9], 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 the Intuitionistic Type Theory is based on constructive logic but not classical logic. There is fundamental difference between them. Interested readers can take a look at [2]. Now, we will first begin with the formalisation of propositional logic.

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 always to be true, the corresponding type must have at least one element.

```
data ⊤ : Set where
  tt : ⊤
```

Falsehood For a proposition always to be false, the corresponding type must have no elements at all.

```
data ⊥ : Set where
```

Conjunction Suppose A and B are propositions, then the proofs of $A \wedge B$ should be consist of a proof of A and a proof of B. In Type Theory, it corresponds to the product type.

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

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

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

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

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

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

Now, we can define the elimination rule of disjunction 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 transform every arbitary proof in A to the falsehood ($\perp$).

```
¬ : Set → Set
¬ A = A → ⊥
```

Implication We say that A implies B if and only if we can transform every proof of A into a proof of B. In Type Theory, it corresponds to a function from A to B, i.e. $A \to B$.

Equivalence Equivalence of two propositions A and B can be seperated into A implies B and B implies A. Therefore, we can consider it as a conjunction of the two implications.

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

Now, we can prove some theorems in propositional logic using the formalisation above, for example, if P implies Q and Q implies R, then P implies R.

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

By completing the function, we have provided an element to the type $(P \to Q) \to (Q \to R) \to (P \to R)$. Therefore, we have also proved that the proposition is true.

Predicate Logic We will now move on to predicate logic and to introduce the universal ($\forall$) and existenial ($\exists$) quantifiers. A predicate is represented by a dependent type in the form of $A \to Set$. For example, we can define the predicate of even numbers and odd numbers inductively as follow:

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

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

Universal Quantifier The interpretaion of the universal quantifier is similar to implication. In order for $\forall x \in A.B(x)$ to be true, we will have to transform every proof a of A into a proof of the predicate $B[x := a]$. 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 zero) = inj₂ base
lem₁ (suc (suc n)) with lem₁ n
... | inj₁ nIsEven = inj₁ (step n nIsEven)
... | inj₂ nIsOdd = inj₂ (step n nIsOdd)
```

Existential Quantifier The interpretation of the existensial quantifier is simliar to conjunction. In order for $\exists x \in A.B(x)$ to be true, we need to provide a proof a of the A and a proof p of the predicate $B[x := a]$. 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 to $\exists[x \in A]B$. Now, we can prove that there exists a natural number which is even.

```
lem₂ : ∃[ n → ℕ ] n isEven
lem₂ = zero , base
```

Decidability A proposition P is decidable if and only if there exists an algorithm that can decide whether it is true or false. We will define the decidability of a proposition as follow:

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

Propositional Equality Another important feature of Type Theory is that we can define the notion of equality in it. 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
```

Note that we can put $refl$ in the proof only because both $suc(suczero)$ and $1 + 1$ are the same 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 y
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
```

2.1.4 Program Specifications as Types

Furthermore, dependent types also allow us to encode program specification within the same platform. Now, let us define a predicate of sorted list (sorted by ascending order). For simplicity, we will only consider the list of natural numbers in here. Before we can define the predicate, we need to have another predicate saying that a given natural number is smaller than every number in a given list.

```
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
```

$\_ \leq \_ : \mathbb{N} \to \mathbb{N} \to Set$ is a binary relation between two natural numbers. Now, we can define an insertion function that takes a natural number and a list as the arguments and returns a list of natural numners. The insertion function is designed so that if the input list is already sorted, then the output list should 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
```

$\_ \leq ?\_ \ : \ \forall n\ m \rightarrow Dec(n \leq m)$ is the decidability of $\_ \leq \_$, 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)
```

The part *Sorted-ASC as* corresponds to the pre-condition and *Sorted-ASC* (*insert n as* corresponds to the post-condition. Once we have completed the function, we will also have proven the specification to be true. Interested readers can try to finish the proof.

2.2 Related Work

Firsov and Uustalu [6] have conducted a similar research on the formalisation of Automata Theory. In their work, they have formalised the translation of regular expressions to NFA together with its correctness proofs in Agda. In their definition of NFA, the set of states and its subsets were represented as vectors. ...

3 Formalisation in Type Theory

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

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

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

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

3.1 Subsets and Decidable Subsets

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

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

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

The two definitions have different purposes. $Subset$ is used to represent $Language$ because not every language is decidable. For other parts such as a subset of states in an automaton, $DecSubset$ is used as the decidability is assumed in the definition. The two definitions are defined in Subset.agda and Subset/DecidableSubset.agda respectively as stated at the top. Operators such as membership ($\in$), subset ($\subseteq$), superset ($\supseteq$) and equality ($=$) can also be found in the two files.

3.1.1 Operations on Subsets

3.1.2 Operations on Decidable Subsets

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

3.2 Languages

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

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

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

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

3.2.1 Operations on Languages

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

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

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

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

3.3 Regular Languages and Regular Expressions

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

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

Listing 1: Regular languages

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

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

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

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

Listing 2: Regular expressions

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

Listing 3: Languages denoted by regular expressions

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

3.4 ϵ-Non-deterministic Finite Automata

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

Definition 4.1 We define Σ^e as the union of Σ and $\{\epsilon\}$, i.e. $\Sigma^e = \Sigma \cup \{\epsilon\}$.

In Agda, this can be expressed by a data type definition:

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

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

For example, the string 'Agda' can be represented by ($\alpha\ A :: \ \alpha\ g :: E :: \ \alpha\ d :: E :: \ \alpha\ a :: []$) or ($E :: \ \alpha\ A :: E :: E :: \ \alpha\ g :: \ \alpha\ d :: E :: \ \alpha\ a :: []$). We say that these two lists are ϵ-strings of the word 'Agda'. When pattern matching on an ϵ-string, we can know if there is an ϵ-transition or not. Other operators and lemmas regarding ϵ-strings such as $to\Sigma^* \ : \ \Sigma^{e*} \rightarrow \Sigma^*$ can also be found in Language.agda.

Now, let us define ϵ-NFA.

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

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

Listing 4: ϵ-NFA

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

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

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

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

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

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

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

```
_⊢ᵏ_─_ : (Q × 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 4.7 We say that $C \vdash^* C'$ if and only if there exists a number of chains n such that $C \vdash^n C'$.

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

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

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

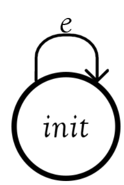
```
Lᵉᴺ : ε-NFA → Language
Lᵉᴺ nfa = λ w →
          ∃[ wᵉ ∈ Σᵉ* ] (w ≡ toΣ* wᵉ × (∃[ q ∈ Q ] (q ∈ᵈ F × (q₀ , wᵉ) ⊢* (q , []))))
```

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

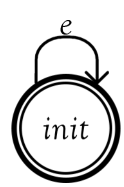
3.5 Thompson's Construction

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

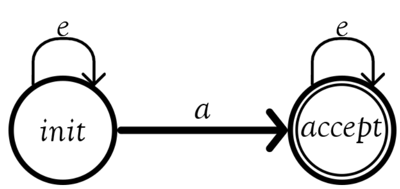
1. for $\emptyset$, we have $M = (\{init\},\ \Sigma^e,\ \delta,\ init,\ \emptyset)$ and graphically



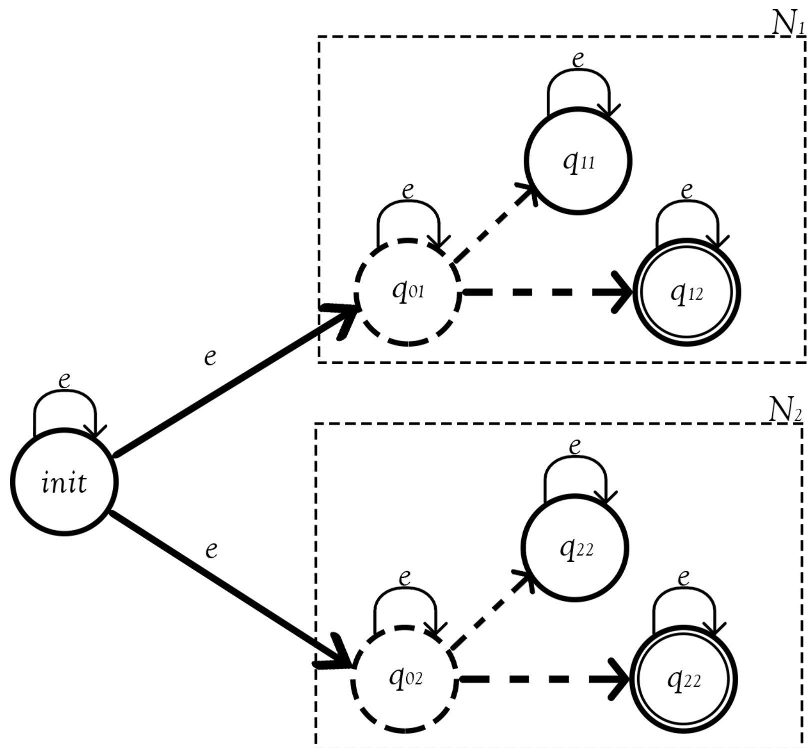
2. for ϵ, we have $M = (\{init\},\ \Sigma^e,\ \delta,\ init,\ \{init\})$ and graphically



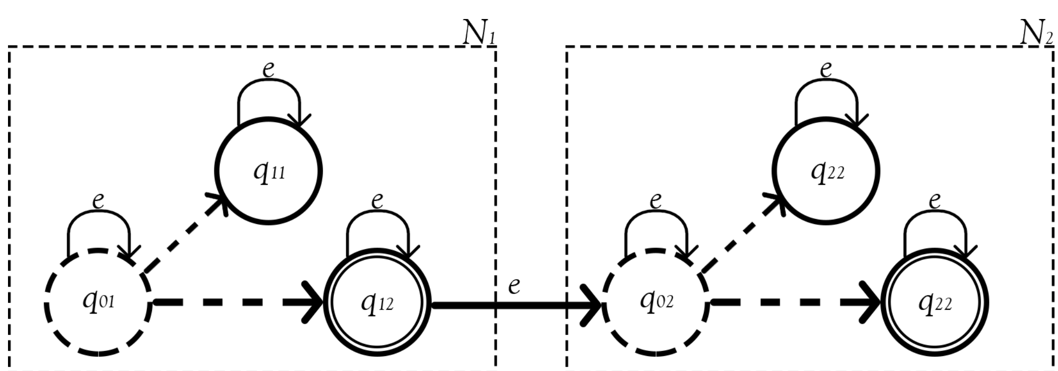
3. for a, we have $M = (\{init, accept\},\ \Sigma^e,\ \delta,\ init,\ \{accept\})$ and graphically



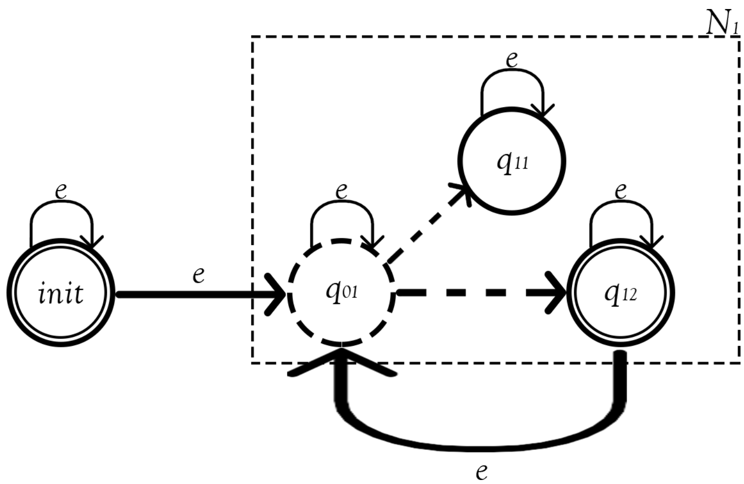
4. if $N_1 = (Q_1,\ \delta_1,\ q_{01},\ F_1)$ and $N_2 = (Q_2,\ \delta_2,\ q_{02},\ F_2)$ are ϵ-NFAs translated from the regular expressions e_1 and e_2 respectively, then
   (a) for $(e_1 \mid e_2)$, we have $M = (\{init\} \cup Q_1 \cup Q_2,\ \Sigma^e,\ \delta,\ init,\ F_1 \cup F_2)$ and graphically



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



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



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

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

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

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

Inductive steps

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

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

1.2) if M accepts w, then there must exists a chain $(init, w^e) \vdash^* (q, \epsilon)$ in M such that w^e is an ϵ-string of w and $q \in F$. Since $q \in F$, therefore $q \neq init$. By Definition 5.1, there are only two possible ways for $init$ to reach q, via q_{01} or ii) q_{02}. Assuming case i), then we have $(init, \epsilon^+ w_1) \vdash^* (q_{01}, w_1)$ and $(q_{01}, w_1) \vdash^* (q, \epsilon)$ where $w^e = \epsilon^+ w_1$ and $q \in Q_1$. Since we have $q \in F$ and $q \in Q_1$; therefore we have $q \in F_1$. Also w_1 is also an ϵ-string of w, thus the chain $(q_{01}, w_1) \vdash^* (q, \epsilon)$ implies that $w \in L(N_1)$. By induction hypothesis, we have $w \in L(e_1)$ and thus $w \in L(e_1 \mid e_2)$. The same argument also applies for case ii). Since we have proved that $w \in L(M) \Rightarrow w \in L(e_1 \mid e_2)$; therefore $L(e_1 \mid e_2) \supseteq L(M)$ also follows;

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

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

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

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

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

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

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

3.2) if M accepts w, ...

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

3.6 Non-deterministic Finite Automata

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

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

Listing 5: NFA

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

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

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

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

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

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

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

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

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

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

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

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

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

3.7 Removing ϵ-transitions

...

3.8 Deterministic Finite Automata

...

3.9 Powerset Construction

...

3.10 Minimal DFA

...

3.11 Minimising DFA

...

4 Further Extensions

Myhill-Nerode Theorem, Pumping Lemma

5 Evaluation

5.1 Correctness and Readability

According to Geuvers [7], a proof has two major roles: 1) to convince the readers that a statement is correct and 2) to expain why a statement is correct. A proof is a verification of a statement which is consist of smaller reasoning steps. These steps will eventually contributes to the correctness of the proof. We can say that the a proof is correct if and only if every reasoning step within the proof is correct. The second part requires the proof to be able to give an intuition of why the statement is correct. In the paragraphs below, we will discuss these two criteria.

Correctness Traditionally, when a mathematician submits the proof of his/her concepts, a group of mathematicians will evaluate and check its correctness. Alternatively, if we formalise the proof in a proof assistant, the proof will be checked automatically by the compiler. The only difference is that we are now relying on the compiler and the machine that runs the compile rather than the group of mathematicians. Therefore, if the compiler and the machine works properly, then any formalised proof that can be compiled without errors are said to be correct. In our case, we have the type checker and the termination checker in Agda to serve the purpose. Furthermore, the correctness of a proof should be obtained by verifiying the individual smaller reasoning steps within the proof. When writing proofs in paper, we usually omit the proofs of some obvious lemmas. However, in Agda, we have provide explicit proofs to every lemma we have used. Therefore, the correctness of an Agda proof will always depend on the correctness of the smaller proofs that it contains.

Readability The other purpose of a proof is to explain why a certain statement is correct. However, in general, a computer proof is incompetent on this purpose.

5.2 Different choices of representation

In this project, we have made several decisions over the representation of mathematical objects. When writing proofs in papers, we are not usually required to provide concreate representations for abstract mathematical objects such as subsets. However, when writing proofs in Type Theory, we usually have to do that. The consequence is that different forms of represenation will lead to different formalisations and thus contributes to the easiness/difficulty in writing the proofs. In the following part, we will discuss different representations we have choosen and their effect.

In our approach, we used $Q : Set$ to represent the set of states while in [6], they used vector as the representation. The

5.3 Problem arises

reachable states problem

6 Conclusion

Recap what is done

Is it feasible to write computer proof in practice?

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendices

Agda Code?