Parsing 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

Copyright © 2016 School of Computer Science, University of Birmingham

Abstract

Parsing Regular Expressions in Agda

Wai Tak, Cheung

Blah blah blah. Blah blah blah.

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

Acknowledgments

Blah blah blah. Blah blah blah.

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 [12]. Automata Theory is a very extensive piece of work, it will be unrealistic to include all 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. Futhermore, this project also gives a brief introduction on how complex and non-trival proofs are formalised in Type Theory.

The Agda formalisation can be separated into two major components: 1) the translation from regular expressions to DFA and 2) the correctness proofs of the translation. In this stage, we are only interested in the correctness of the translation but not the efficiency.

1.1 Motivation

...

1.2 Overview

In section two, we will first give a brief introduction on Agda as a programming language and as a proof assistant. Then we will define the types that are frequently used in the project. We will also look into some small Agda proofs so that readers can have a taste of how proofs are formalised in Type Theory. In the end of section two, we will discuss a similar reseach [6] conducted by Firsov and Uustalu. Following the background, the third section will be a detail description of our work. We will walk through the Agda formalisation of the two components stated in previous part. After that, we will introduce some possible extensions to the project. Then, there will an evaluation. 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 [10] at the Chalmers University of Technology. In this section, we will describe the basic features of Agda and how dependent types are employed to build programs and proofs. Most of the materials presented below can also be found in the two tutorial papers [3] and [11]. Interested readers can take a look at them and get a more precise idea on how to work with Agda. We will begin by showing how to do ordinary functional programming in Agda.

2.1.1 Simply Typed Functional Programming

Haskell is the implementaion language of Agda, as we will see below, Agda has borrowed many features from Haskell. In below, we will show how to define basic data types (boolean and natural number) and functions over them.

Boolean We first introduce 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* since they take no argument. Their types are explicitly declared in ': *Bool*'. On the other hand, *Bool* is a member of the type *Set*. *Set* represents the set of all *small types*. *Bool* is a *small type* but *Set* itself is not, it is a *large type*. We will look into the difference in later part. Now, let us define the negation function on boolean values:

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

Unlike Haskell, we must provide a type declaration for every functions. 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 The type of natural numbers is defined 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 as *suc zero*. Now, let us define the addition over natural numbers 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 inductively as:

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

2.1.2 Universe

...

2.1.3 Dependent Types

Dependent types are types that depend on values of other types. For example, A^n is the type of vectors with length n (depends on n). These kinds of types are not possible to be declared in simply-typed systems like Haskell or Ocaml. Now, let us look at how it is defined using dependent type in Agda.

```
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 $(A : Set) : \mathbb{N} \rightarrow Set$ *where*, $A : Set$ is a type parameter defining the type of elements in the vector. While the part $\mathbb{N} \rightarrow Set$ means that Vec takes a value n of type $\mathbb{N}$ and produce a vector type of A that depends on n. For example, $Vec\ A\ zero$ is the type of vectors with zero length.

With dependent types, we can define data types and functions which are more expressive and precise. For instance, we can define the function *head* which returns the first element in a vector as follow:

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

Unlike in Haskell which we need to pattern match on [] and produce an error, we only need to pattern match on the case $(x :: xs)$ here 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 such that the order of elements in the tree is guranteed by the type declaration. However, we will not be looking into that as it is not our major concern. Interersted readers can take a look at Section 6 in [3]. Other than data type declaration, dependent types are also used to encode predicate logic and program specifications. We will look at these two applications in later parts in this section.

2.1.4 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-Lof 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; therefore, in this project, we are working with constructive logic rather than classical logic. Interested readers can take a look at [2].

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 an element in the set; it is false if and only if its set is empty.

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

```
data ⊤ : Set where
  tt : ⊤
```

Falsehood For a proposition to be false, the corresponding type must have no element 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 corresponds to the introduction rule of conjunction while the elimination rules correspond to the functions fst and snd:

```
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 be consist of either a proof of A or a proof of B. In Type Theory, it corresponds to the sum type.

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

Here is the function corresponds to the elimination rule of disjunction:

```
⊎-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 negation is defined as a function that transform any arbitary proof in A to the falsehood. It is defined as follow:

```
¬ : Set → Set
¬ A = A → ⊥
```

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

Predicate Logic We will now move to predicate logic and introduce the universal ($\forall$) and existenial ($\exists$) quantifiers.

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 proofs a of A to a proof of the proposition $B[x := a]$. In Type Theory, it is represented by the function $(x : A) \to B\ x$.

Existential Quantifier ...

Propositional Equality ...

2.1.5 Encoding Program Specifications

...

2.2 Related Work

Certified Parsing of Regular Languages. The matrix representation.

3 Formalisation in Type Theory

Let us recall the two objectives of the project: 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. $Subset\ A = A \to Set$.

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

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

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

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

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 $\_\hat{}\_$ is defined recursively as:

```
_^_ : Language → Language → Language
L ^ zero    = ⟦ϵ⟧
L ^ (suc n) = L • L ^ 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
  ∅   : 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 ∅ = ∅
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 $List\ \Sigma^*$. However, this definition gives us no way to extract an ϵ-transition from the input string. Therefore, we need to introduce another representation of the set of strings specifically for this purpose. (For Definition 4.1 and 4.2, please refers to Language.agda)

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

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

```
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 $to\Sigma^* : \Sigma^{e*} \to \Sigma^*$ 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 × Σ^e × Σ^e*) → (Q × Σ^e*) → Set
(q , a , w) ⊢ (q' , w') = w ≡ w' × q' ∈^d δ q a
```

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

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

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

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

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

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

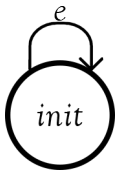
```
L^eN : ϵ–NFA → Language
L^eN nfa = λ w →
           ∃[ w^e ∈ Σ^e* ] (w ≡ toΣ* w^e × (∃[ q ∈ Q ] (q ∈^d F × (q0 , w^e) ⊢* (q , []))))
```

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

3.5 Thompson's Construction

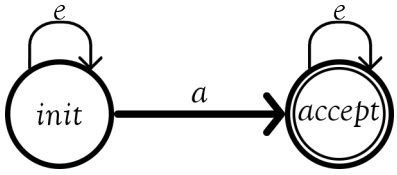
Definition 5.1 The translation for any regular expressions to an ϵ-NFA is defined inductively as follow:

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



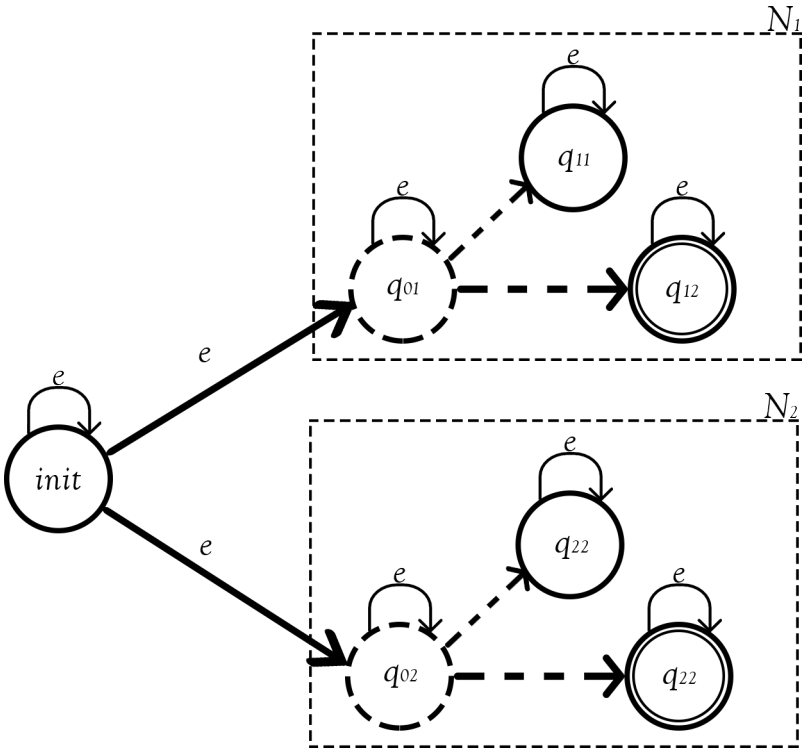
2. for ϵ, we have $M = (\{init\},\ \Sigma^e,\ \delta,\ init,\ \{init\})$ and graphically

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

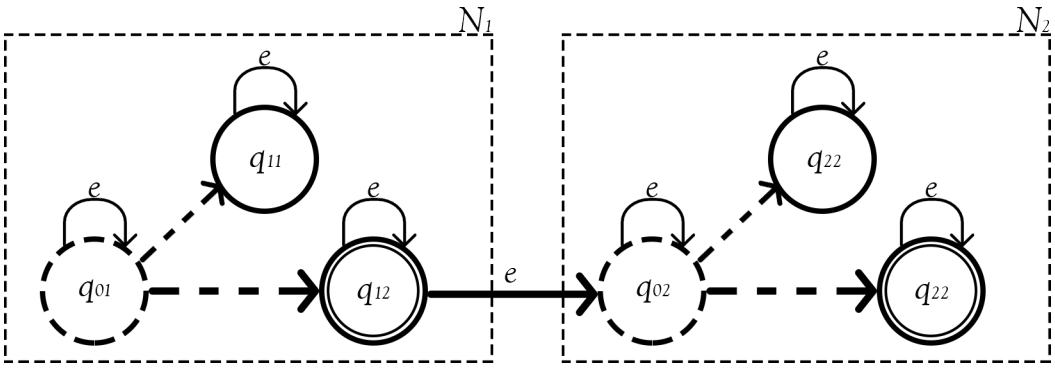


4. if $N_1 = (Q_1,\ \delta_1,\ q_{01},\ F_1)$ and $N_2 = (Q_2,\ \delta_2,\ q_{02},\ F_2)$ are ϵ-NFAs translated from the regular expressions e_1 and e_2 respectively, then

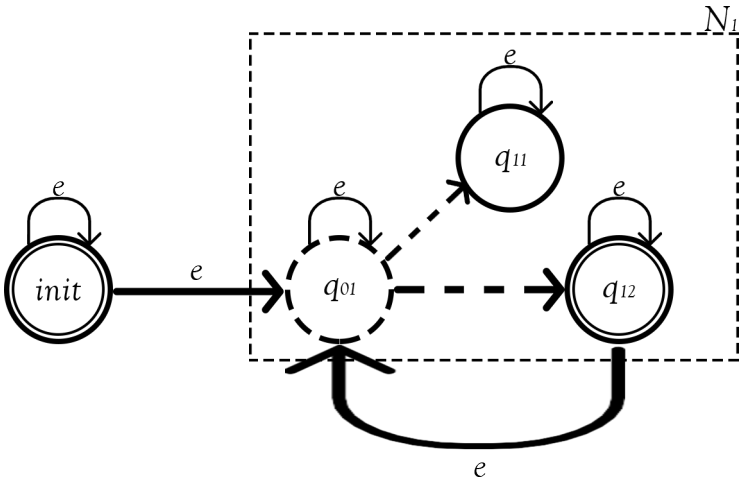
   (a) for $(e_1 \mid e_2)$, we have $M = (\{init\} \cup Q_1 \cup Q_2,\ \Sigma^e,\ \delta,\ init,\ F_1 \cup F_2)$ and graphically



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

```
L^N : NFA → Language
L^N 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 [7], proofs have two major roles: 1) to convince the readers that the statement is correct and 2) to expain why the statement is correct. The first part is consist of the verification of small individual reasoning steps. We have to determines if these steps constitute a correct proof. The second part requires the proof to be able to give an intuition of the statement. In the paragraphs below, we will evaluate the project based on these two criteria.

Correctness In the traditional way, when a mathematician submits the proof of his/her concepts, a group of mathematicians will evaluate it and see if the proof is correct or not. 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 it runs on rather than a 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 termination checker in Agda to serve the purpose. Another aspect is that the correctness of a proof should be obtained by verifiying the individual small reasoning steps within the proof. When writing proofs in paper, we usually omit the proofs of some obvious theorem such as the associativity of addition. However, in Agda, we have to make explicit all the proofs for every lemma and theorem. Therefore, the correctness of a Agda proof will always depend on the correctness of the smaller proofs that it contains. We can conclude that a computer proof is a very convincing argument.

Readability The readability of a proof means how good it is at explaining why the statement is correct.

5.2 Different Choices of Representaion

Apart from the two criteria, the other major aspect on the evaluation of a computer proof is the way that the proof is formalised. When writing proofs in papers, we are usually not required to provide concreate representations for abstract mathematical objects like subsets. However, when writing proofs in Type Theory, we usually have to provide a representation for them. 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 the different representations we have choosen and their consequences.

The set of states (Q) in an automata ...

Subset and DecidableSubset ...

6 Discussion

If start over ... ?

Is it feasible to write computer proof in practice?

7 Conclusion

Recap what is done

References

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

[8] William A. Howard. The formulae-as-types notion of construction, 1969.

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

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

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

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

Appendices

Agda Code?