UNIVERSITÀ DEGLI STUDI DI TORINO

DIPARTIMENTO DI INFORMATICA

SCUOLA DI SCIENZE DELLA NATURA

Laurea Triennale in Informatica

Formalization of Regular Languages in Agda

Relatore: Prof. Luca Padovani

Candidato: Desislav Nikolaev Ivanov

ANNO ACCADEMICO
2019 / 2020

Abstract

We use the proof assistant Agda to prove properties on Regular Expressions and Finite State Automata. Initially, we prove the correctness of two ways to decide whether or not a string belongs to the language of a regular expression. The first one uses Brzozowski's Derivatives and the second one transforms regular expressions into Non-deterministic Finite State Automata. Finally, we show the pumping lemma for regular languages and conclude with the non-regularity of an example language.

Contents

Chapter 1

Introduction to Agda

Agda is a dependently typed functional programming language. Its typesystem is so powerful that it can be used as a proof assistant. Checking if a function is correct can be done by successfully compiling the file containing it.

1.1 Inductive datatypes

Datatypes can be defined inductively. For example, the type for natural numbers has a base case zero and an inductive case for the successor. In Agda, these two cases are defined as constructors:

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

`zero` is the constructor for the base case and `suc` receives a number as paramether and constructs its successor. For example, one is defined as `suc zero`, two is defined as `suc (suc zero)` and three is defined as `suc (suc (suc zero))`.
Writing big numbers with this notation can be tedious, so Agda provides a shortcut for built-in types such as numbers:

```
{-# BUILTIN NATURAL ℕ #-}

y : ℕ
y = 123
```

1.2 Functions

We can define the addition operation recursively:

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

Here `_+_` is the name of the function. It receives two natural numbers as parameters and produces a natural number. Underscores are used to define infix notation and indicate where the arguments should be placed. In this case, the sum of two numbers `n`, `m`, can be written as `n + m`, which is equivalent to not using infix notation at all `(_+_ n m)`.
The definition uses pattern matching to match all possible constructors for the first argument. It has a base case for `zero` and an inductive case for the successor. In the first case, adding `zero` to a number `n` returns simply `n`. In the second case, adding `suc m` to `n` returns the successor of the sum of the two smaller ones `suc (m + n)`.
When defining functions, Agda performs termination checks. A function terminates correctly when the recursion is done on smaller arguments.

1.3 Induction

Inductive datatypes are somehow related to the (structural) induction principle. Proving a property for an inductive datatype can be done by showing that the property holds for the base cases and then showing it holds for the inductive cases, eventually using the inductive hypothesis.
In Agda, the induction hypothesis is generated by recursion.
Here is an example which proves the right identity of addition:

```
+-identityʳ : ∀ (m : ℕ) → m + zero ≡ m
+-identityʳ zero = refl
+-identityʳ (suc m) with +-identityʳ m
... | IH = cong suc IH
```

For the base case we have `m = zero` and so the goal requires a proof for `zero + zero ≡ zero`. By definition of addition, `zero + zero = zero` and we conclude `zero ≡ zero`. For the inductive case, we need to prove `suc (m + zero) ≡ suc m`. The `with` command evaluates the induction hypothesis using recursion on `m`. The value of `IH` is `m + zero ≡ m`. We conclude the case using the congruence property of equality, which states that if two arguments are equal, then equality is preserved after applying the same function to both:

```
cong : ∀{A C : Set} {a : A} {b : A}
  (f : A → C)
  → a ≡ b
  → f a ≡ f b
cong f refl = refl
```

The proof for identity can be simplified with the `rewrite` command. It takes an equality argument like $a \equiv b$ and it internally instructs Agda to rewrite b where it finds a. The rewriting step is done only once, in order to avoid infinitely growing rewrites:

```
+-identity^r' : ∀ (m : ℕ) → m + zero ≡ m
+-identity^r' zero = refl
+-identity^r' (suc m) with +-identity^r m
... | IH rewrite IH = refl
```

The inductive case requires a proof for `suc (m + zero) ≡ suc m`.
Using the `rewrite` command on the induction hypothesis (same as the previous one: `m + zero ≡ m`), Agda internally updates the left side of the goal from `suc (m + zero)` to `suc (m)`. The required proof becomes `suc m ≡ suc m` which is immediate.
For these proofs, we actually used another datatype: the equality datatype. It is a dependent datatype, meaning that it depends on a concrete value:

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

1.4 Dependent datatypes

Dependent datatypes are types depending on one or more values. For example, the type for finite-size vectors can be defined as a dependent type:

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

It depends on a natural number, the size of the vector. The `[]` constructor builds a vector of size `zero`. The `_::_` constructor takes two parameters, an element of type `A` and a vector of size `n`, and builds a vector of size `1 + n`.
Here is an example of a vector containing two natural numbers:

```
x : Vec ℕ 2
x = 1 :: 3 :: []
```

We can also define the type for finite numbers `Fin n`:

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

This means that a value of type `Fin n` can only be a number between `zero` and `n - 1`. For example, only the numbers `0` and `1` can be of type `Fin 2`:

```
a : Fin 2
a = zero

b : Fin 2
b = suc zero
```

Saying that `2` is of type `Fin 2` is wrong and Agda rejects it:

```
c : Fin 2
c = suc (suc zero)
-- Error :
-- (suc _n_34) != zero of type ℕ
-- when checking that the expression zero has type Fin 0
```

Using `Fin` and `Vec` types, we can define a total function for accessing a value at an index in a vector:

```
lookup : ∀ {n} → Vec A n → Fin n → A
lookup (x :: xs) zero    = x
lookup (x :: xs) (suc i) = lookup xs i
```

Since lookup is total, there cannot be any run-time error while accessing a value at a given index (e.g. IndexOutOfBoundsException in Java). If we were to write `x = lookup (1 :: []) (suc zero)`, the type checker would raise an error as the vector is of type `Vec ℕ 1`, whereas the index is of type `Fin 2`.

1.4.1 Propositions as types

By taking advantage of dependent datatypes, logical propositions can be defined as types. For example, the following type can be used to constructively build a proposition stating that a number is Even:

```
data Even : ℕ → Set where
  zero : Even zero
  next : ∀{n : ℕ}
    → Even n
    → Even (suc (suc n))
```

We say that `0` is even and adding `2` to an even number constructs another even number. We can define a similar proposition for Odd numbers where the base case is `1` instead of `0`:

```
data Odd : ℕ → Set where
  one : Odd (suc zero)
  next : ∀{n : ℕ}
    → Odd n
    → Odd (suc (suc n))
```

Using these two propositions, we can prove that for an arbitrary number `n`, `Even n` implies `Odd (suc n)`:

```
even⇒suc-odd : ∀{x : ℕ}
  → Even x
    -----------
  → Odd (suc x)
even⇒suc-odd zero = one
even⇒suc-odd (next a) = next (even⇒suc-odd a)
```

Here we used the function type as an implication.
We can also say that `Odd n` implies `Even (suc n)`:

```
odd⇒suc-even : ∀{x : ℕ} → Odd x → Even (suc x)
odd⇒suc-even one = next zero
odd⇒suc-even (next a) = next (odd⇒suc-even a)
```

We can also define connectives as datatypes. For example, conjunction can be defined as the product type (also interpretable as a Pair) and disjunction can be defined as the sum type (also referred to as the Either type):

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

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

For example, we can prove that every natural number is even or odd:

```
ℕ-Even-or-Odd : ∀(x : ℕ) → Even x ⊎ Odd x
ℕ-Even-or-Odd zero = inj₁ zero
ℕ-Even-or-Odd (suc x) with ℕ-Even-or-Odd x
... | inj₁ ev = inj₂ (even⇒suc-odd ev)
... | inj₂ od = inj₁ (odd⇒suc-even od)
```

Regarding true and false, we can still be constructive. The type for truth is the unit type, with a single constructor `tt`. On the other hand, false is the empty type, without any constructor:

```
data ⊤ : Set where
  tt : ⊤

data ⊥ : Set where
-- no constructors
```

False implies anything. In fact, Agda automatically closes the case as no constructor pattern that can be matched:

```
⊥-elim : ∀ {A : Set}
  → ⊥
  → A
⊥-elim ()
```

Negation is a function which takes a type `A` and creates a function which takes an argument of the type `A` and produces the empty type:

```
¬_ : Set → Set
¬ A = A → ⊥
```

Given a proposition `¬ A`, if we provide an argument of type `A`, we are able to construct the empty type, which is absurd as empty has no constructors. Therefore, we could conclude anything using `⊥-elim`. For example, we can show that if a number is even, then it is not odd. As a consequence, we can conclude that no natural number is both even and odd:

```
even⇒¬odd : ∀{x : ℕ} → Even x → ¬ Odd x
even⇒¬odd zero = λ ()
even⇒¬odd (next ev) (next od) = ⊥-elim((even⇒¬odd ev) od)

ℕ-not-both-even-odd : ∀{x : ℕ} → ¬ (Even x × Odd x)
ℕ-not-both-even-odd ⟨ ev , od ⟩ = ⊥-elim (even⇒¬odd ev od)
```

The “if and only if” proposition is a record with two fields, each representing one direction of the implications:

```
record _⇔_ (A B : Set) : Set where
  field
    to   : A → B
    from : B → A
```

Isomorphism between types is defined similarly to “if and only if”, with two additional fields which provide the proofs about the identity of compositions:

```
record _≃_ (A B : Set) : Set where
  field
    to   : A → B
    from : B → A
    from∘to : ∀ (x : A) → from (to x) ≡ x
    to∘from : ∀ (y : B) → to (from y) ≡ y
```

Propositional types provide constructive evidence why propositions hold, but we can also define propositions using the Bool type and define connectives as functions:

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

_∧_ : Bool → Bool → Bool
true  ∧ true  = true
_     ∧ _     = false

_∨_ : Bool → Bool → Bool
true  ∨ _      = true
_     ∨ true   = true
false ∨ false  = false

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

The Bool type can be mapped to the relative ⊤ and ⊥ propositions:

```
T : Bool → Set
T true   =  ⊤
T false  =  ⊥
```

A function that produces boolean values can include both true and false cases, but it provides less information than the propositional types approach. On the other hand, the latter must be defined manually by providing a construction (in fact, we cannot prove some laws like the "excluded middle law" `∀{A : Set} → A ⊎ ¬ A`).
Both benefits can be included in one type, the decidable type. It combines the functional aspect where true and false can be decided, and it also concretely provides constructive proofs:

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

For example, for any natural number `n`, we can decide whether the proposition `Even n` holds or not:

```
Even? : ∀(x : ℕ) → Dec (Even x)
Even? x with ℕ-Even-or-Odd x
... | inj₁ xEven = yes xEven
... | inj₂ xOdd = no (λ xEven → ℕ-not-both-even-odd ⟨ xEven , xOdd ⟩)
```

We use the property `ℕ-Even-or-Odd` which states that a number `x` is even or odd (the case where both propositions hold is not excluded).
If `x` is even, we conclude with `yes` by providing the same proof determined by the lemma. Otherwise, when `x` is odd, we conclude that `x` is not even. In fact, we show that assuming `x` is also even allows us to construct the empty type by applying `ℕ-not-both-even-odd`.

1.4.2 Vec, Fin and Subset properties

In this section, we define the `Subset` type which we use later. A Subset of size n is a vector of size n containing Bool values.

```
Subset : ℕ → Set
Subset n = Vec Bool n
```

An element `i : Fin n` is in the subset if the value at index `i` in the vector is `true`.

```
_∈_ : ∀{n} → (p : Fin n) → (ss : Subset n) → Set
p ∈ ss = T(lookup ss p)

_∉_ : ∀{n} → (p : Fin n) → (ss : Subset n) → Set
p ∉ ss = ¬ T(lookup ss p)
```

We can decide whether or not an element belongs to a subset:

```
_∈?_ : ∀{n} → (p : Fin n) → (ss : Subset n) → Dec (p ∈ ss)
zero  ∈? (false ∷ z) = no (λ z → z)
zero  ∈? (true ∷ z)  = yes tt
suc p ∈? (x ∷ z)     = p ∈? z
```

A `Fin n` value can be transformed into a natural number:

```
toℕ : ∀ {n} → Fin n → ℕ
toℕ zero    = 0
toℕ (suc i) = suc (toℕ i)
```

A `Fin m` value can be raised arbitrarily into `Fin (n + m)`:

```
raise : ∀ {m} n → Fin m → Fin (n ℕ.+ m)
raise zero    i = i
raise (suc n) i = suc (raise n i)
```

A `Fin m` value can also be injected into `Fin (m + n)`:

```
inject+ : ∀ {m} n → Fin m → Fin (m ℕ.+ n)
inject+ n zero    = zero
inject+ n (suc i) = suc (inject+ n i)
```

Given a `Fin (n + m)` value, we can split the sum generating either `Fin n` or `Fin m`:

```
splitAt : ∀ m {n} → Fin (m + n) → Fin m ⊎ Fin n
splitAt zero    i        = inj₂ i
splitAt (suc m) fzero    = inj₁ fzero
splitAt (suc m) (fsuc i) = Data.Sum.map fsuc (λ x → x) (splitAt m i)
```

And here are two proofs about raising, injecting and splitting:

```
splitAt-raise : ∀ m n i → splitAt m (raise {n} m i) ≡ inj₂ i
splitAt-raise zero    n i = refl
splitAt-raise (suc m) n i
  rewrite splitAt-raise m n i = refl

splitAt-inject+ : ∀ m n i → splitAt m (inject+ n i) ≡ inj₁ i
splitAt-inject+ (suc m) n fzero = refl
splitAt-inject+ (suc m) n (fsuc i)
  rewrite splitAt-inject+ m n i = refl
```

Chapter 2

Regular Expressions

2.1 Definition

Regular expressions can be defined recursively based on an alphabet Σ. The language of a regular expression R is denoted as $L(R)$.
Base cases:

- $\varnothing$ is a regular expression and $L(\varnothing)$ is $\varnothing$, the empty language
- ϵ is a regular expression and $L(\epsilon)$ is $\{\epsilon\}$, the language containing just the empty string
- c, where c is an element of the alphabet Σ, is a regular expression and $L(c)$ is $\{c\}$, the language containing the single-character string c

Induction:

- $R + S$ is a regular expression denoting $L(R) \cup L(S)$, where R and S are regular expressions
- $R \cdot S$ is a regular expression denoting $L(R)L(S)$, where R and S are regular expressions
- $R*$ is a regular expression denoting $(L(R))*$, where R is a regular expression

2.2 Implementation

In Agda, we first define the regular expression type:

```
module Regexp (Σ : Set) where
ϵ = []
infixl 6 _+_
infixl 7 _·_
infixl 8 _⋆
data RegExp : Set where
  ⟨⟩   : RegExp
  ⟨ϵ⟩   : RegExp
  Atom : Σ ⟶ RegExp
  _+_  : RegExp ⟶ RegExp ⟶ RegExp
  _·_  : RegExp ⟶ RegExp ⟶ RegExp
  _⋆   : RegExp ⟶ RegExp
```

It has six constructors:

- $<>$ is the constructor for the regular expression denoting the empty language
- $<\epsilon>$ is the constructor for the regular expression denoting the language containing the empty string
- $Atom$ receives an element of the alphabet as a parameter and builds a regular expression denoting the language containing the single-character string described by the parameter
- $\_+\_$ receives two regular expressions as parameters and builds the regular expression denoting the union of the two languages denoted by the parameters
- $\_\cdot\_$ receives two regular expressions as parameters and builds the regular expression denoting the concatenation of the language of the left-side regular expression to the language of the right-side regular expression
- $\_*$ receives a regular expression as a parameter and builds the regular expression accepting the concatenation of zero or indefinitely more strings of the language of the parameter

For example, fixing the alphabet to $\Sigma = \{a, b, c\}$, we can define the following regular expressions:

- $(aa)*$, denoting the language with even number of as
- $a * b?a*$, denoting the strings of as and bs with at most one b
- $a * b*$, denoting the strings of as followed by bs

- $(a(c+b))*$, denoting the strings of pairs of a followed by either c or b

```
[aa]⋆ = (Atom a · Atom a) ⋆
a⋆b?a⋆ = Atom a ⋆ · (Atom b + ⟨ϵ⟩) · Atom a ⋆
a⋆b⋆ = Atom a ⋆ · Atom b ⋆
[a[c+b]]⋆ = (Atom a · (Atom c + Atom b)) ⋆
```

Our current definition of regular expressions is only syntactic and does not hold any information about their languages. We define the semantics of the languages as an inductive relation between strings and regular expressions:

```
data _∈_ : String → RegExp → Set where
  in-ϵ  : ϵ ∈ ⟨ϵ⟩
  in-⋆1 : ∀ {E : RegExp}
          → ϵ ∈ (E ⋆)
  in-c  : (c : Σ) → (c ∷ ϵ) ∈ Atom c
  in-·  : ∀ {s t : String} {E F : RegExp}
          → s ∈ E
          → t ∈ F
          → (s ++ t) ∈ (E · F)
  in+l  : ∀ {s : String} {E F : RegExp}
          → s ∈ E
          → s ∈ (E + F)
  in+r  : ∀ {s : String} {E F : RegExp}
          → s ∈ F
          → s ∈ (E + F)
  in-⋆2 : ∀ {s t : String} {E : RegExp}
          → s ∈ E
          → t ∈ (E ⋆)
          → (s ++ t) ∈ (E ⋆)
```

There are three base cases:

- the empty string ϵ belongs to $L(\epsilon)$
- for any regular expression E, the empty string ϵ belongs to $L(E*)$
- given an element `c` of the alphabet Σ, the single-character string `c` belongs to the language of the regular expression `Atom c`

Induction:

- for any s, t, E, F, if s belongs to $L(E)$ and if t belongs to $L(F)$ then the concatenation of the strings st belongs to $L(E \cdot F)$
- given s and E such that s belongs to $L(E)$, s belongs to $L(E + F)$ for any regular expression F
- given s and F such that s belongs to $L(F)$, s belongs to $L(E + F)$ for any regular expression E
- given s, t and E, if s belongs to $L(E)$ and if t belongs to $L(E*)$, then st belongs to $L(E*)$

Referring to the regular expressions previously defined, we can show some examples of constructive definitions of membership relations.
The regular expression `(aa)⋆` matches `aaaa`, and so the relation $aaaa \in L((aa)*)$ can be defined as follows:

```
x : (a ∷ a ∷ a ∷ a ∷ []) ∈ [aa]⋆
x = in-⋆2 (in-· (in-c a) (in-c a))
        (in-⋆2 (in-· (in-c a) (in-c a)) in-⋆1)
```

For `a⋆b?a⋆`, we can show that the relation $aba \in L(a * b?a*)$ holds:

```
y : (a ∷ b ∷ a ∷ []) ∈ a⋆b?a⋆
y = in-· (in-· (in-⋆2 (in-c a) in-⋆1)
               (in+l (in-c b)))
         (in-⋆2 (in-c a) in-⋆1)
```

For `a⋆b⋆`, we can show that $aabbb \in L(a * b*)$:

```
z : (a ∷ a ∷ b ∷ b ∷ b ∷ []) ∈ a⋆b⋆
z = in-· (in-⋆2 (in-c a)
            (in-⋆2 (in-c a) in-⋆1))
         (in-⋆2 (in-c b)
            (in-⋆2 (in-c b)
              (in-⋆2 (in-c b) in-⋆1)))
```

And for our last regular expression `(a(c + b))⋆`, we can show that $abac \in L((a(c + b))*)$

```
v : (a ∷ b ∷ a ∷ c ∷ []) ∈ [a[c+b]]⋆
v = in-⋆2 (in-· (in-c a) (in+r (in-c b)))
          (in-⋆2
            (in-· (in-c a) (in+l (in-c c))) in-⋆1)
```

We can already prove some properties on our membership predicate. For example, given a relation `s ∈ (E · F)`, we can show that `s` can be divided into two strings `t`, `u`, such that `t ∈ E` and `u ∈ F`:

```
split-seq : ∀{s E F}
  → s ∈ (E · F)
  → ∃[ u ] ∃[ v ] ((s ≡ u ++ v) × (u ∈ E) × (v ∈ F))
split-seq (in-· p q) = _ , _ , refl , p , q
```

We can prove a similar property for the star operation:

```
split-⋆ : ∀{E s}
  → s ∈ (E ⋆)
  → s ≢ ϵ
  → ∃[ u ] ∃[ v ] (u ≢ ϵ × s ≡ u ++ v × u ∈ E × v ∈ (E ⋆))
split-⋆ in-⋆1 q = ⊥-elim (q refl)
split-⋆ (in-⋆2 {[]} p q) neps = split-⋆ q neps
split-⋆ (in-⋆2 {x ∷ s} {t} p q) _ = x ∷ s , t , (λ ()) , refl , p , q
```

A consequence of `split-seq` is the fact that if the empty string belongs to the concatenation of two regular expression, then the empty string belongs to the languages of both regular expressions.

```
ϵ-seq : ∀{E F} → ϵ ∈ (E · F) → ϵ ∈ E × ϵ ∈ F
ϵ-seq p with split-seq p
... | [] , [] , refl , p1 , p2 = p1 , p2
```

2.3 Algebraic laws for regular expressions

Two regular expressions are equivalent if they denote the same language. For example, given two regular expressions a and b, the regular expressions $a + b$ and $b + a$ denote the same language. This is also known as the commutativity law of $+$. To prove it we use isomorphism between types. We show that for an arbitrary string s and regular expressions E and F, the relation $s \in (E + F)$ is isomorphic to the relation $s \in (F + E)$:

```
+-comm : ∀ {s : String} {E F : RegExp}
  → s ∈(E + F) ≃ s ∈(F + E)
+-comm {s} {E} {F} =
  record
    { to      = to
    ; from    = from
    ; from∘to = from∘to
    ; to∘from = to∘from
    }
  where
    to : s ∈(E + F) → s ∈(F + E)
    to (in+l x) = in+r x
    to (in+r x) = in+l x

    from : s ∈(F + E) → s ∈(E + F)
    from (in+l x) = in+r x
    from (in+r x) = in+l x

    from∘to : (x : s ∈ (E + F)) → from (to x) ≡ x
    from∘to (in+l x) = refl
    from∘to (in+r x) = refl

    to∘from : (x : s ∈ (F + E)) → to (from x) ≡ x
    to∘from (in+l x) = refl
    to∘from (in+r x) = refl
```

Another example is the fact that the regular expression $E(F+G)$ is equivalent to $(EF+EG)$. This is also known as the distributive law of $+$ over concatenation:

```
seq-distrib-+ˡ : ∀ {s : String} {E F G : RegExp}
  → s ∈(E · (F + G)) ≃ s ∈(E · F + E · G)
seq-distrib-+ˡ {s} {E} {F} {G} =
  record
    { to      = to
    ; from    = from
    ; from∘to = from∘to
    ; to∘from = to∘from }
  where
    to : s ∈ (E · (F + G)) → s ∈ (E · F + E · G)
    to (in-· x (in+l y)) = in+l (in-· x y)
    to (in-· x (in+r y)) = in+r (in-· x y)

    from : s ∈ (E · F + E · G) → s ∈ (E · (F + G))
    from (in+l (in-· x y)) = in-· x (in+l y)
    from (in+r (in-· x y)) = in-· x (in+r y)

    from∘to : (x : s ∈ (E · (F + G))) → from (to x) ≡ x
    from∘to (in−· x (in+l y)) = refl
    from∘to (in−· x (in+r y)) = refl

    to∘from : (y : s ∈ (E · F + E · G)) → to (from y) ≡ y
    to∘from (in+l (in−· y y1)) = refl
    to∘from (in+r (in−· y y1)) = refl
```

In our work, we proved many other algebraic laws, such as associativity of $+$, identity of concatenation and idempotency of $*$, which can be found in our Agda files.

2.4 Decidable membership using derivatives

To determine whether or not a string belongs to the language denoted by a regular expression, one cannot simply list and compare all the strings of the language until he finds a match, as they may be infinite (e.g. $L(a*)$). To solve this problem we show an algorithm which makes use of a concept of derivatives on regular expressions, also known as Brzozowski's Derivatives. Later we show another way to solve the problem, by transforming regular expressions into finite state automata.

2.4.1 Nullable predicate

Initially we define the Nullable predicate. A regular expression E is Nullable when $\epsilon \in L(E)$. We proceed with an inductive datatype:

```
data Nullable : RegExp → Set where
  null⟨ϵ⟩ : Nullable ⟨ϵ⟩
  null+l : ∀{F G} → Nullable F → Nullable (F + G)
  null+r : ∀{F G} → Nullable G → Nullable (F + G)
  null·  : ∀{F G} → Nullable F → Nullable G → Nullable (F · G)
  null⋆  : ∀{F} → Nullable (F ⋆)
```

The empty string belongs to the regular expressions $\langle\epsilon\rangle$ and F⋆, for any F. If a regular expression F contains ϵ, then its union with any other regular expression contains ϵ. If ϵ belongs to two regular expressions F and G, then it belongs to their concatenation.
All the regular expressions defined previously satisfy the Nullable predicate, whereas, for instance, Atom c does not:

```
x : Nullable [aa]⋆
x = null⋆
```

```
y : Nullable a⋆b?a⋆
y = null· (null· null⋆ (null+r null⟨ϵ⟩)) null⋆

z : Nullable a⋆b⋆
z = null· null⋆ null⋆

u : Nullable [a[c+b]]⋆
u = null⋆

v : ¬ Nullable (Atom a)
v = λ ()
```

Is our `Nullable` predicate actually correct? We need to prove that ϵ belongs to the language of a regular expression `E` if and only if `E` is `Nullable`. Here is the proof:

```
theorem1 : ∀{E : RegExp}
  → ϵ ∈(E) ⇔ Nullable E
theorem1 = record { to = to ; from = from }
  where
    to : ∀{E} → ϵ ∈ E → Nullable E
    to {⟨ϵ⟩} _         = null⟨ϵ⟩
    to {E + F} (in+l x) = null+l (to x)
    to {E + F} (in+r x) = null+r (to x)
    to {E · F} x with ϵ-seq x
    ... | ϵ∈E , ϵ∈F     = null· (to ϵ∈E) (to ϵ∈F)
    to {E ⋆} _          = null⋆

    from : ∀{E} → Nullable E → ϵ ∈ E
    from null⟨ϵ⟩     = in-ϵ
    from (null+l x)  = in+l (from x)
    from (null+r x)  = in+r (from x)
    from (null· x y) = in-· (from x) (from y)
    from null⋆       = in-⋆1
```

The `Nullable` predicate is decidable, meaning that we can always know if a regular expression is `Nullable` or not, and as a consequence if its language contains the empty string or not. We define this property as a function which takes a regular expression `E` and produces `Dec (Nullable E)`:

```
Nullable? : (E : RegExp) → Dec (Nullable E)
Nullable? ⟨⟩ = no (λ ())
Nullable? ⟨ϵ⟩ = yes null⟨ϵ⟩
```

```
Nullable? (Atom c) = no (λ ())
Nullable? (r + s) with Nullable? r | Nullable? s
... | yes p | _     = yes (null+l p)
... | _     | yes q = yes (null+r q)
... | no ¬p | no ¬q = no λ{ (null+l p) → ⊥-elim (¬p p)
                         ; (null+r q) → ⊥-elim (¬q q) }
Nullable? (r · s)  with Nullable? r | Nullable? s
... | yes p | yes q = yes (null· p q)
... | _     | no ¬q = no λ{ (null· _ q) → ⊥-elim (¬q q) }
... | no ¬p | _     = no λ{ (null· p _) → ⊥-elim (¬p p) }
Nullable? (r ⋆) = yes null⋆
```

2.4.2 Derivative

The derivative of a regular expression E with respect to an element a of the alphabet is defined by induction on the structure of E, denoted as $E[a]$:

```
_[_] : RegExp → Σ → RegExp
⟨⟩ [ a ]  = ⟨⟩
⟨ϵ⟩ [ a ] = ⟨⟩
(Atom b)[ a ] with b ≟ a
... | yes p  = ⟨ϵ⟩
... | no ¬p  = ⟨⟩
(F + G)[ a ] = F [ a ] + G [ a ]
(F · G)[ a ] with Nullable? F
... | yes p = F [ a ] · G + G [ a ]
... | no ¬p = F [ a ] · G
(F ⋆)[ a ]  = F [ a ] · F ⋆
```

For example, the derivative of (aa)⋆ with the symbol a, denoted as (aa)⋆[a], is equal to (aa)[a] (aa)⋆, which becomes a[a] a(aa)⋆ and gets reduced to ϵa(aa)⋆. If we use b instead of a, we get that (aa)⋆[b] = (aa)[b] (aa)⋆ = a[b] a(aa)⋆ = ⟨⟩a(aa)⋆. Since ⟨⟩ is the annihilator of concatenation, we can show that no strings belong to this last derivative:

```
x : [aa]⋆ [ a ] ≡ ⟨ϵ⟩ · Atom a · [aa]⋆
x = refl

y : [aa]⋆ [ b ] ≡ ⟨⟩ · Atom a · [aa]⋆
y = refl
```

```
yp : ∀{s} → ¬ s ∈ [aa]⋆ [ b ]
yp (in-· (in-· () _) _)
```

Given a regular expression E, a symbol a and a string v,

$$av \in L(E) \Leftrightarrow v \in L(E[a]).$$

For example, the string `aaaa` belongs to the language of `(aa)⋆` and the string `aaa` belongs to the language of `(aa)⋆[a]`:

```
x1 : (a ∷ a ∷ a ∷ a ∷ []) ∈ [aa]⋆
x1 = in-⋆2 (in-· (in-c a) (in-c a))
         (in-⋆2 (in-· (in-c a) (in-c a)) in-⋆1)

x2 : (a ∷ a ∷ a ∷ []) ∈ ([aa]⋆ [ a ])
x2 = in-· (in-· in-ϵ (in-c a))
         (in-⋆2 (in-· (in-c a) (in-c a)) in-⋆1)
```

We prove this property by induction on the membership predicate, in both directions. We also use the previously proven properties `theorem1`, `split-seq` and `split-⋆`:

```
theorem2 : ∀{a : Σ} {v : String} {E : RegExp}
  → v ∈(E [ a ]) ⇔ (a ∷ v) ∈(E)
theorem2 = record { to = to ; from = from }
  where
    to : ∀{a v E} → v ∈(E [ a ]) → (a ∷ v) ∈( E )
    to {a} {v} {Atom c} x with c ≟ a
    to {_} {[]} {Atom c} x | yes refl = in-c c
    to {E = F + G} (in+l x) = in+l (to x)
    to {E = F + G} (in+r x) = in+r (to x)
    to {E = F · G} x with Nullable? F
    to {E = F · G} (in+l (in-· x y)) | yes p
      = in-· (to x) y
    to {E = F · G} (in+r x)          | yes p
      = in-· (_⇔_.from theorem1 p) (to x)
    to {E = F · G} (in-· x y)        | no ¬p
      = in-· (to x) y
    to {E = F ⋆} (in-· x y) = in-⋆2 (to x) y

    from : ∀ {a}{v}{E} → (a ∷ v) ∈ E → v ∈ E [ a ]
    from {_} {_} {Atom c} (in-c .c) with c ≟ c
    ... | yes p = in-ϵ
    ... | no ¬p = ⊥-elim (¬p refl)
    from {E = F + G} (in+l x) = in+l (from x)
    from {E = F + G} (in+r x) = in+r (from x)
    from {E = F · G} x with Nullable? F | split-seq x
    ... | yes p | [] , av , refl , _ , av∈G
      = in+r (from av∈G)
    ... | yes p | a ∷ u , t , refl , au∈F , t∈G
      = in+l (in-· (from au∈F) t∈G)
    ... | no ¬p | [] , _ , refl , ϵ∈F , _
      = ⊥-elim (¬p (_⇔_.to theorem1 ϵ∈F))
    ... | no ¬p | a ∷ u , t , refl , au∈F , t∈G
      = in-· (from au∈F) t∈G
    from {E = F ⋆} x with split-⋆ x (λ ())
    ... | [] , _ , ¬p , _ , _ , _ = ⊥-elim (¬p refl)
    ... | a ∷ t , v , _ , refl , at∈E , v∈E⋆
      = in-· (from at∈E) v∈E⋆
```

2.4.3 Decidable membership

We can show that a string $v = a_1 a_2 \cdots a_n$ belongs to the language of a regular expression F if and only if we obtain a Nullable regular expression after applying derivation on F with each symbol in v:

$$v \in L(F) \Leftrightarrow \text{Nullable}\ (F[a_1][a_2] \cdots [a_n])$$

The proof is by induction on v, as an immediate consequence of theorem1 and theorem2 :

```
theorem3 : ∀ {v : String} {F : RegExp}
  → v ∈(F) ⇔ Nullable (foldl _[_] F v)
theorem3 = record { to = to ; from = from }
  where
    to : ∀ {v} {F} → v ∈(F) → Nullable (foldl _[_] F v)
    to {[]} x      = _⇔_.to theorem1 x
    to {v ∷ vs} x = to (_⇔_.from theorem2 x)

    from : ∀ {v} {F} → Nullable (foldl _[_] F v) → v ∈ F
    from {[]} x      = _⇔_.from theorem1 x
    from {v ∷ vs} x = _⇔_.to theorem2 (from x)
```

The Nullable predicate is decidable and both foldl and derivation are executable functions. Therefore, given an arbitrary string v and an arbitrary regular expression F, we can use `theorem3`, `Nullable?`, `foldl` and `_[_]` to check if the membership relation `v ∈ F` holds:

```
_∈?_ : (v : String) → (F : RegExp) → Dec (v ∈ F)
v ∈? F with Nullable? (foldl _[_] F v)
... | yes p = yes (_⇔_.from theorem3 p)
... | no ¬p = no (λ z → ¬p (_⇔_.to theorem3 z))
```

In other words, we can check if a string belongs to the language of a regular expression. Here is an example:

```
v = (a :: b :: a :: a :: a :: a :: a :: a :: a :: a :: [])
e1 = v ∈? a⋆b?a⋆
e2 = (b :: v) ∈? a⋆b?a⋆
```

By evaluating `e1`, Agda determines that `v` belongs to the language of `a⋆b?a⋆` and provides the following construction:

```
yes
  (in-· (in-· (in-⋆2 (in-c a) in-⋆1) (in+l (in-c b)))
      (in-⋆2 (in-c a)
       (in-⋆2 (in-c a)
        (in-⋆2 (in-c a)
         (in-⋆2 (in-c a)
          (in-⋆2 (in-c a)
           (in-⋆2 (in-c a)
            (in-⋆2 (in-c a)
             (in-⋆2 (in-c a) in-⋆1)))))))))
```

On the other hand, as expected, the evaluation of `e2` determines that the membership relation does not hold, since the regular expression allows only one *b*.

Chapter 3

Finite State Automata

In this chapter we introduce deterministic finite state automata (DFAs) and non-deterministic finite state automata (NFAs). We emulate the inductive operations of regular expressions (concatenation, union, star) using NFAs and we use them to recursively transform regular expressions into NFAs which recognize the same languages. In the last section, we prove the pumping lemma for DFAs using the pigeonhole principle and provide an example of a non-regular language.

3.1 Deterministic Finite Automata

A deterministic finite state automaton is a 5-tuple, $(Q, \Sigma, \delta, S, F)$, consisting of:

- a finite set of states Q
- a finite set of input symbols called the alphabet Σ
- a transition function $\delta : Q \times \Sigma \to Q$
- a starting state $S \in Q$
- a subset of final states $F \subseteq Q$

To enforce the finite property, in Agda we define DFAs as dependent records based on a natural number parameter. Our states are of type `Fin n`. The starting state is a `Fin n` element, the transition function is defined as `Fin n → Σ → Fin n` and the final states subset is `Subset n`:

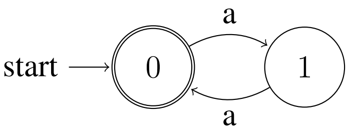
```
module Dfa (Σ : Set) where

record Dfa (n : ℕ) : Set where
  field
    S : Fin n
    δ : Fin n → Σ → Fin n
    F : Subset n
```

We define two functions that will simplify the construction of DFAs. The first one is `make-δ` which builds transition functions. It takes two arguments: the error state and a list of triples that describe the transitions. The other function we will use is `make-dfa`. It constructs a DFA based on a number of states, a starting state, an error state, a list of final states and a list of transitions:

```
TransitionsList = λ n → List (Fin n × Σ × Fin n)

make-δ : ∀{n}
    → Fin n
    → TransitionsList n
    → (Fin n → Σ → Fin n)
make-δ err [] = λ _ _ → err
make-δ err ((q , x , p) ∷ xs)
    = λ h y → if ⌊ q ≟ᶠ h ⌋ ∧ ⌊ x ≟ y ⌋
                then p
                else make-δ err xs h y

make-dfa : (n : ℕ)
    → Fin n
    → Fin n
    → List (Fin n)
    → TransitionsList n
    → Dfa n
make-dfa n start error finals transitions
    = record
      { S = start
      ; δ = make-δ error transitions
      ; F = ⋃ (map ⟦_⟧ finals)
      }
```

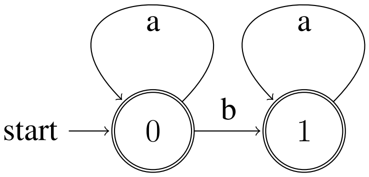
We can now declaratively build DFAs.
For example, we can implement the DFAs which recognize the same languages of the

regular expressions previously used. As a convention, all undefined transitions end up in the error state and are not shown in the diagrams. For `(aa)*`, we can use three states $\{0, 1, 2\}$, where 0 is the starting state and 2 is the error state:



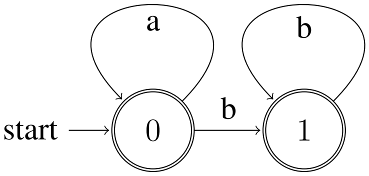
```
dfa-[aa]* = make-dfa 3 0F 2F (0F :: []) (
        (0F , a , 1F)
      :: (1F , a , 0F)
      :: []
    )
```

The regular expression `a*b?a*` can also be defined using three states, where 0 is the starting state and 2 is the error state:



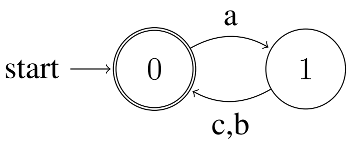
```
dfa-a*b?a* = make-dfa 3 0F 2F (0F :: 1F :: []) (
        (0F , a , 0F)
      :: (0F , b , 1F)
      :: (1F , a , 1F)
      :: []
    )
```

The implementation for `a*b*` is very similar to the previous one:



```
dfa-a*b* = make-dfa 3 0F 2F (0F :: 1F :: []) (
        (0F , a , 0F)
      :: (0F , b , 1F)
      :: (1F , b , 1F)
      :: []
    )
```

For (a(c+b))* we again use three states:



```
dfa-[a[c+b]]* = make-dfa 3 0F 2F (0F :: []) (
        (0F , a , 1F)
      :: (1F , b , 0F)
      :: (1F , c , 0F)
      :: []
    )
```

3.1.1 Extended transition function

The language of a DFA is the set of strings for which the computation from the starting state ends up in a final state. We can express the computation as a function, also known as the extended transition function. It receives a DFA as input, a state, and a string and recursively computes the resulting state based the definition of the δ function of the DFA:

```
open Dfa

δ^ : ∀{n} → (dfa : Dfa n) → (q : Fin n) → String → Fin n
δ^ dfa q [] = q
δ^ dfa q (x :: s) = δ^ dfa (δ dfa q x) s
```

Here is a simple property about the extended transition function applied to the concatenation of two strings:

```
lemma-δ^ : ∀{n}
  (dfa : Dfa n)
  → (s : String)
  → (t : String)
  → (q : Fin n)
  → δ^ dfa q (s ++ t) ≡ δ^ dfa (δ^ dfa q s) t
lemma-δ^ dfa [] t q = refl
lemma-δ^ dfa (c :: s) t q = lemma-δ^ dfa s t (δ dfa q c)
```

3.1.2 Language of a DFA

A string s belongs to the language of a DFA A, here denoted as A_s, if the extended transition function leads to a final state, beginning at the starting state of A. Since the result of

the extended transition function is of type `Fin n` and the subset of final states is of type `Subset n`, we define the membership relation between strings and DFAs as the membership relation between `Fin n` and `Subset n`.
As shown previously, the membership relation between `Fin n` and `Subset n` is decidable. Therefore, membership between strings and languages of DFAs is also decidable.

```
infix 10 _↓_
_↓_ : ∀{n} → Dfa n → String → Set
dfa ↓ s  = δ^ dfa (S dfa) s ∈ F dfa

_↓?_ : ∀{n} → (dfa : Dfa n) → (s : String) → Dec (dfa ↓ s)
dfa ↓? s = δ^ dfa (S dfa) s ∈? F dfa
```

We can immediately tell if a string is accepted or rejected by simply evaluating the result of the extended transition function. For example, the DFA for the regular expression `(aa)*` accepts `aa` and rejects string `aaa`:

```
p0 : dfa-[aa]* ↓ (a :: a :: [])
p0 = tt

p1 : ¬ dfa-[aa]* ↓ (a :: a :: a :: [])
p1 ()
```

Another example is the fact that the DFA for `a*b?a*` accepts `aba` and rejects `bba`:

```
p2 : dfa-a*b?a* ↓ (a :: b :: a :: [])
p2 = tt

p3 : ¬ dfa-a*b?a* ↓ (b :: b :: a :: [])
p3 ()
```

3.2 Non-deterministic Finite Automata

Every regular expression can be transformed into a finite state automaton that recognizes the same language (the inverse is also true). Except for the base cases, there are three basic language operations upon which regular expressions are defined: union, concatenation, and star. Implementing these operations using DFAs is rather difficult, so we introduce a new concept of finite state automata: Non-deterministic finite state automata. NFAs might seem more powerful than DFAs, but it can be shown that they are equivalent. In our implementation, we won't use spontaneous transitions, as they could require additional termination proofs.

3.2.1 Definition

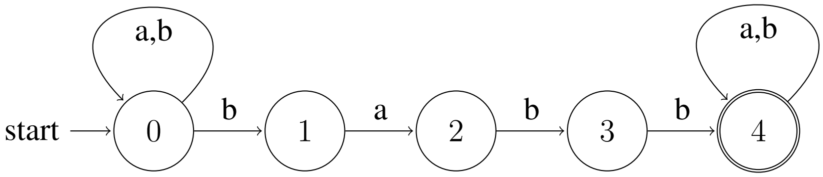
The NFA type is similar to the DFA one, except for the transition function which produces a `Subset n` instead of a single state:

```
record Nfa (n : ℕ) : Set where
  field
    S : Fin n
    δ : Fin n → Σ → Subset n
    F : Subset n
```

To simplify the constructions, we define the helper functions `make-nfa-δ` and `make-nfa` similarly to the ones used for DFAs, with the addition that the third element of the transitions triples is a list of reached states instead of a single state:

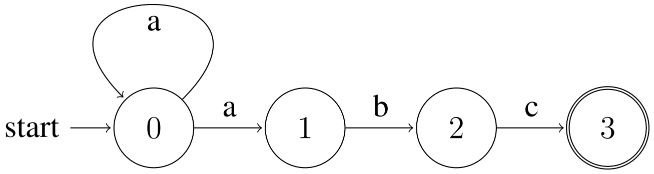
```
NfaTransitionsList = λ n → List (Fin n × Σ × List (Fin n))

make-nfa-δ : ∀{n}
  → Fin n
  → NfaTransitionsList n
  → (Fin n → Σ → Subset n)
make-nfa-δ err [] = λ _ _ → ⟦ err ⟧
make-nfa-δ err ((q , x , ps) ∷ xs)
  = λ h y → if ⌊ q ≟ᶠ h ⌋ ∧ ⌊ x ≟ y ⌋
            then ⋃ (map ⟦_⟧ ps)
            else make-nfa-δ err xs h y

make-nfa : (n : ℕ)
  → Fin n
  → Fin n
  → List (Fin n)
  → NfaTransitionsList n
  → Nfa n
make-nfa n start error finals transitions
  = record
    { S = start
    ; δ = make-nfa-δ error transitions
    ; F = ⋃ (map ⟦_⟧ finals)
    }
```

For example, we can implement a NFA accepting the strings of as and bs containing the substring $babb$.



```
nfa-babb-substring = make-nfa 6 0F 5F (4F ∷ []) (
        (0F , a , 0F ∷ [])
      ∷ (0F , b , 0F ∷ 1F ∷ [])
      ∷ (1F , a , 2F ∷ [])
      ∷ (2F , b , 3F ∷ [])
      ∷ (3F , b , 4F ∷ [])
      ∷ (4F , a , 4F ∷ [])
      ∷ (4F , b , 4F ∷ [])
      ∷ []
    )
```

Another example is the NFA accepting as terminating by abc:



```
nfa-term-by-abc = make-nfa 5 0F 4F (3F ∷ []) (
        (0F , a , 0F ∷ 1F ∷ [])
      ∷ (1F , b , 2F ∷ [])
      ∷ (2F , c , 3F ∷ [])
      ∷ []
    )
```

3.2.2 Language

The `Subset n` result of the δ adds the non-determinism. The computation of a NFA is no longer linear, as it proceeds on all the ramifications of the resulting subsets of δ. A string is accepted if the NFA reaches a final state in any of the branches:

```
open Nfa
accepts : ∀{n} → Nfa n → Fin n → String → Bool
accepts A q []       = F A ! q
accepts A q (c :: s)
  = any λ p → (δ A q c) ! p ∧ accepts A p s
```

Where any checks if any number of type Fin n satisfies a given function:

```
any : ∀{n} → (P : Fin n → Bool) → Bool
any {zero}  P = false
any {suc _} P = P fzero ∨ any λ x → P (fsuc x)
```

A string belongs to the language of a NFA A if A accepts the string beginning at its starting state:

```
_↓_ : ∀{n} → Nfa n → String → Set
A ↓ s  = T (accepts A (S A) s)
```

Since accepts is a simple function, the membership relation A ↓ s becomes easily decidable by evaluating the result of accepts:

```
_↓?_ : ∀{n} → (A : Nfa n) → (s : String) → Dec (A ↓ s)
A ↓? s = T? (accepts A (S A) s)
```

For example, referring to the NFA nfa-babb-substring, we can show that it accepts the strings *babb* and *ababbbb*, but rejects the string *aba*:

```
babb : String
babb = b :: a :: b :: b :: []

x : nfa-babb-substring ↓ babb
x = tt

y : nfa-babb-substring ↓ (a :: babb ++ b :: b :: [])
y = tt

z : ¬ nfa-babb-substring ↓ (a :: b :: a :: [])
z ()
```

Another example is the fact that nfa-term-by-abc accepts the strings *abc* and *aaaabc* but rejects the string *bac*:

```
abc : String
abc = a :: b :: c :: []

t : nfa-term-by-abc ↓ (abc)
```

```
t = tt

u : nfa-term-by-abc ↓ (a :: a :: a :: abc)
u = tt

v : ¬ nfa-term-by-abc ↓ (b :: a :: c :: [])
v ()
```

When a NFA A accepts a non-empty string like $x :: xs$ from a state q, we can compute one "next" state reached from q with symbol x, from which A accepts xs:

```
-- Lemma
anyToExists : ∀{n} {f : Fin n → Bool}
  → T (any f)
  → ∃[ i ] T(f i)

nextState : ∀{n x xs q} {nfa : Nfa n}
  → T (accepts nfa q (x :: xs))
  → ∃[ p ] ( p ∈ (δ nfa q x)  × T (accepts nfa p xs))
nextState {n} acc-x-xs with anyToExists {n} acc-x-xs
... | p , p∈δqx_∧_acc-p-xs = p , (splitAnd p∈δqx_∧_acc-p-xs)
```

Saying that a NFA accepts the empty string is equivalent to saying that its starting state is final:

```
emptyLanguage : ∀{n} {nfa : Nfa n}
    → nfa ↓ ϵ ≡ S nfa ∈ F nfa
emptyLanguage = refl
```

3.2.3 Union

Given two NFAs A and B, we can build a new NFA C whose language is the union of the two: $L(C) = L(A) \cup L(B)$. The number of states of the resulting NFA is the sum of the number of states of the original two, with one additional state. This additional state is also the starting state. Given a state q and a symbol c, the transition function $\delta(q, c)$ is defined in the following way:

- case q is zero (the additional starting state): the resulting subset contains the states reached by A and B from their starting states with the given input symbol c
- case q is a state of A: the resulting subset contains the states reached by A from q with c

- case q is a state of B: the resulting subset contains the states reached by B from q with c

The final states subset contains the final states of A and B and eventually the new starting state if the empty string belongs to A or B.
Given two subsets T of size n and U of size m, using the vector concatenation function, we can create a subset of size $n + m$ containing distinctly all the elements of T and U.
We use the previously described function `splitAt` to determine if a given state of type `Fin (1 + n + m)` is the additional state, a state of A or a state of B.

```
union : ∀{n m} → Nfa n → Nfa m → Nfa (1 + n + m)
union {n} {m} A B =
  record
    { S = fzero
    ; δ = d
    ; F = sf ++ (F A) ++ (F B)
    }
  where
    d : Fin (1 + n + m) → Σ → Subset (1 + n + m)
    d q c  with splitAt 1 q
    d q c | inj₁ z = ∅ {1} ++ (δ A (S A) c) ++ (δ B (S B) c)
    d q c | inj₂ f with splitAt n f
    ... | inj₁ l = ∅ {1} ++ (δ A l c) ++ ∅
    ... | inj₂ r = ∅ {1} ++ ∅ ++ (δ B r c)

    sf : Subset 1
    sf with A ↓? ϵ | B ↓? ϵ
    sf | no ϵ∉l | no ϵ∉r = ∅
    sf | _      | _      = FullSet
```

For example, we can build the union of the two NFAs previously introduced and we can show it accepts *abc* and *babb* but rejects *bac*:

```
nfa-union-babb-abc = union nfa-babb-substring nfa-term-by-abc

q : nfa-union-babb-abc ↓ (abc)
q = tt

r : nfa-union-babb-abc ↓ (b :: a :: b :: b :: [])
r = tt
```

```
s : ¬ nfa-union-babb-abc ↓ (b ∷ a ∷ c ∷ [])
s ()
```

Before proving that the construction is correct, we first need to prove that the two NFAs are still present in the new NFA after the construction. We show this by proving that both NFAs individually accept a string from a state if and only if they accept it from the same state relatively injected/raised into the union.
The proofs are by induction on the string. For the NFA provided as first argument, the relative state is injected:

```
union-preservesˡ : ∀{n m} {A : Nfa n} {B : Nfa m} {q : Fin n}
  → (s : String)
  → T (accepts A q s)
    ⇔
    T (accepts (union A B) (fsuc (inject+ m q)) s)
union-preservesˡ {n}{m} {A} {B} {q} s =
  record { to = to s q ; from = from s q }
  where
  to : (s : String) (q : Fin n)
    → T (accepts A q s)
    → T (accepts (union A B) (fsuc (inject+ m q)) s)
  to [] q acc with
      A ↓? ϵ
    | B ↓? ϵ
    | ++-inject (F A) (F B) q acc
  ...| yes _ | yes _ | v = v
  ...| yes _ | no  _ | v = v
  ...| no  _ | yes _ | v = v
  ...| no  _ | no  _ | v = v
  to (c ∷ s) q acc with nextState {_} {c} {s} acc
  ... | w , w∈δqc , t rewrite splitAt-inject+ n m q
                  with to s w t
                  | ++-inject {n}{m} (δ A q c) ∅ w w∈δqc
  ... | z | d = fromExists (inject+ m w , (joinAnd d z))

  from : (s : String) (q : Fin n)
    → T (accepts (union A B) (fsuc (inject+ m q)) s)
    → T (accepts A q s)
  from [] q d rewrite
        sym (lookup-++ˡ (F A) (F B) q)
    with A ↓? ϵ | B ↓? ϵ
  ... | yes _ | yes _  = d
  ... | yes _ | no  _  = d
  ... | no  _ | yes _  = d
  ... | no  _ | no  _  = d
  from (x :: s) q d with nextState {_} {x} {s} {fsuc (inject+ m q)} d
  ... | w , w∈δqx , accWs rewrite splitAt-inject+ n m q
    with q∈ss++∅ w (δ A q x) w∈δqx
  ... | p , eq , ds rewrite eq =
    fromExists (p , joinAnd ds (from s p accWs))
```

For the NFA provided as second argument, the proof is essentially the same and we only list its statement:

```
union-preserves^r : ∀{n m} {A : Nfa n} {B : Nfa m} {q : Fin m}
  → (s : String)
  → T (accepts B q s)
    ⇔
    T (accepts (union A B) (fsuc (raise n q)) s)
```

We now show that a string belongs to the language of the union if and only if it belongs to the language of the first or the second NFA.
The proof is by induction on the string as a consequence of union-preserves^l and union-preserves^r:

```
union-correct : ∀{n m : ℕ} {A : Nfa n} {B : Nfa m}
  → (s : String)
  → A ↓ s ⊎ B ↓ s ⇔ union A B ↓ s
union-correct {n}{m}{A}{B} s =
  record { to = to s ; from = from s }
  where
  to : (s : String) → A ↓ s ⊎ B ↓ s → union A B ↓ s
  to [] ac with A ↓? ϵ | B ↓? ϵ
  ... | yes _  | yes _ = tt
  ... | yes _  | no  _ = tt
  ... | no  _  | yes _ = tt
  ... | no ¬p  | no ¬q = ⊥-elim ((¬p ¬-⊎ ¬q) ac)

  to (c :: s) (inj_1 A↓cs) with nextState {_} {c} {s} A↓cs
  ... | w , w∈δA , accepts-s-w-A
                with ++-inject (δ A (S A) c) (δ B (S B) c) w w∈δA
                   | _⇔_.to (union-preserves^l s) accepts-s-w-A
  ... | w∈δA∪B | accepts-s-w-A∪B
```

```
 = fromExists (inject+ m w , joinAnd w∈δAUB accepts-s-w-AUB)

to (c ∷ s) (inj_2 B↓cs) with nextState {_} {c} {s} B↓cs
... | w , w∈δB , accepts-w-B
       with ++-raise (δ A (S A) c) (δ B (S B) c) w w∈δB
        | _⇔_.to (union-preserves^r s) accepts-w-B
... | w∈δAUB | accepts-s-w-AUB
 = fromExists (raise n w , joinAnd w∈δAUB accepts-s-w-AUB)


from : (s : String) → union A B ↓ s → A ↓ s ⊎ B ↓ s
from [] d with A ↓? ϵ | B ↓? ϵ
... | yes p | yes p_1 = inj_1 p
... | yes p | no ¬p  = inj_1 p
... | no ¬p | yes p  = inj_2 p

from (x ∷ s) d with nextState {suc n + m} {x} {s} {0F} d
... | w , w∈δSc , accepts-w-s
 with split-∈++ w (δ A (S A) x) (δ B (S B) x) w∈δSc
... | inj_1 (p , eq , z) rewrite eq =
 inj_1 (fromExists (
  p , joinAnd z (_⇔_.from (union-preserves^l s) accepts-w-s)))
... | inj_2 (p , eq , z) rewrite eq =
 inj_2 (fromExists (
  p , joinAnd z (_⇔_.from (union-preserves^r s) accepts-w-s)))
```

3.2.4 Concatenation

The concatenation function is similar to the union. We add a new starting state and build the transition function based on the transition functions of the input NFAs.
Given q and c, $\delta(q, c)$ is defined as follows:

- case q is zero (the additional starting state):
  - if $\epsilon \notin L(A)$: the resulting subset contains only states reached by A from its starting state with symbol c
  - if $\epsilon \in L(A)$: the resulting subset contains the states reached by both NFA from their starting state with symbol c
- case q is a state of A:

  - if q is not final: the resulting subset is the subset of states reached by A from q with the symbol c
  - otherwise: in addition to the previous case subset we also add the states reached by B from its starting state with the symbol c

- case q is a state of B: the result is the subset of states reached by B from q with the symbol c

The final states subset is defined as:

- if ϵ belongs to neither $L(A)$ nor $L(B)$: only the final states of B are final
- if ϵ belongs to both $L(A)$ and $L(B)$: the starting state is final along with the final states of A and B
- if ϵ belong to $L(B)$ but does not belong to $L(A)$: only the final states of A and B are final

And here is the implementation in Agda:

```
concat : ∀{n m} → Nfa n → Nfa m → Nfa (1 + n + m)
concat {n} {m} A B =
  record
    { S = fzero
    ; δ = d
    ; F = f
    }
  where
    d : Fin (1 + n + m) → Σ → Subset (1 + n + m)
    d q c with splitAt 1 q
    d _ c | inj₁ z with A ↓? ϵ
    ... | yes isf = ∅ {1} ++ (δ A (S A) c) ++ (δ B (S B) c)
    ... | no ¬isf = ∅ {1} ++ (δ A (S A) c) ++ ∅
    d _ c | inj₂ mn with splitAt n mn
    d _ c | inj₂ mn | inj₁ l with l ∈? F A
    ... | yes isf = ∅ {1} ++ (δ A l c) ++ (δ B (S B) c)
    ... | no ¬isf = ∅ {1} ++ (δ A l c) ++  ∅
    d _ c | inj₂ mn | inj₂ r = ∅ {1} ++ ∅ ++ (δ B r c)

    f : Subset (1 + n + m)
    f with A ↓? ϵ | B ↓? ϵ
    f | yes ϵ∈l | yes ϵ∈r = FullSet {1} ++ F A ++ F B
    f | no ϵ∉l | yes ϵ∈r = Ø {1} ++ F A ++ F B
    f | _ | no ϵ∉r = Ø {1} ++ Ø ++ F B
```

For example, we can apply this function to our two NFAs and concatenate `nfa-babb-substring` on the left of `nfa-term-by-abc`. We can show that the concatenation accepts `babbabc` but rejects `babb`, because the NFA `nfa-term-by-abc` is on the right and requires a string terminating by `abc`:

```
nfa-concat-babb-abc = concat nfa-babb-substring nfa-term-by-abc

o : nfa-concat-babb-abc ↓ (babb ++ abc)
o = tt

p : ¬ nfa-concat-babb-abc ↓ babb
p ()
```

The correctness proofs follow the same techniques used for the union. We first show that the two NFAs are preserved in the concatenation after the construction. Then we conclude that a string s belongs to the concatenation if and only if it can be divided into two strings u, v such that u belongs to $L(A)$ and v belongs to $L(B)$. Here are the statements of these properties:

```
concat-preservesʳ : ∀{n m : ℕ} {p} {A : Nfa n} {B : Nfa m}
  → (v : String)
  → T(accepts B p v)
    ⇔
    T(accepts (concat A B) (raise 1 (raise n p)) v)

concat-preservesˡ : ∀{n m : ℕ} {q} {A : Nfa n} {B : Nfa m}
  → (s : String)
  → T(accepts (concat A B) (fsuc (inject+ m q)) s)
    ⇔
    ∃[ u ] ∃[ v ] (s ≡ u ++ v × T(accepts A q u) × B ↓ v)

concat-correct : ∀{n m : ℕ} {A : Nfa n} {B : Nfa m}
  → (s : String)
  → concat A B ↓ s
    ⇔
    ∃[ u ] ∃[ v ] (s ≡ u ++ v × A ↓ u × B ↓ v)
```

3.2.5 Star

Given a NFA A with n states, we can define the star operation by adding one single state. It is the new starting state and it is also final, as the empty string always belongs to the star closure. The transition function $\delta(q, c)$ is defined as follows:

- if q is the additional state: the result is the subset of states reached by A from its starting state with symbol c
- otherwise, q is one of the states of A:
  - if q is final: the resulting subset contains the states reached by A with symbol c from both q and its starting state
  - otherwise: the result is simply the subset of states reached by A from q with c

```
star : ∀{n} → Nfa n → Nfa (1 + n)
star {n} nfa =
  record { S = fzero ; δ = d ; F = ⟦ fzero ⟧ ++ F nfa }
  where
    d : Fin (1 + n) → Σ → Subset (1 + n)
    d q c with splitAt 1 q
    d _ c | inj₁ z = ∅ ++ δ nfa (S nfa) c
    d _ c | inj₂ p with p ∈? F nfa
    ... | yes isf = ∅ ++ (δ nfa (S nfa) c) ∪ (δ nfa p c)
    ... | no ¬isf = ∅ ++ δ nfa p c
```

For example, we can show that the `star` construction of `nfa-term-by-abc` accepts the strings *abcabc* and *abcaaabc* but rejects *abccc*:

```
k : nfa-star-term-abc ↓ (abc ++ abc)
k = tt

l : nfa-star-term-abc ↓ (abc ++ a :: a :: abc)
l = tt

m : ¬ nfa-star-term-abc ↓ (abc ++ c :: c :: [])
m ()
```

Again, we first show that the original NFA is preserved after the star construction and then conclude with the correctness. However, in this case, we have two separate proofs, as the empty string always belongs to the star construction but might not be accepted by the original NFA. Here are the statements:

```
star-preserves : ∀{n} {nfa} {q : Fin n} {s}
  → T(accepts (star nfa) (fsuc q) s)
    ⇔
    ∃[ u ] ∃[ v ] (
          s ≡ u ++ v
        × T(accepts nfa q u)
        × star nfa ↓ v)

star-correct1 : ∀{n} {s v : String} {nfa : Nfa n}
  → nfa ↓ s × (star nfa) ↓ v
  → (star nfa) ↓ (s ++ v)

star-correct2 : ∀{n} {a} {s : String} {nfa : Nfa n}
  → (star nfa) ↓ (a :: s)
  → ∃[ u ] ∃[ v ](
          s ≡ u ++ v
        × nfa ↓ (a :: u)
        × star nfa ↓ v)
```

3.3 Deciding RegExp membership using NFAs

3.3.1 NFAs for $L(\varnothing)$, $L(\{\epsilon\})$, $L(\{c\})$

Before diving into the generation of a NFA from a regular expression, we first define the NFAs for the base cases: $L(\varnothing)$, $L(\{\epsilon\})$, $L(\{c\})$.
For the empty language, we use one single state which is the starting state and it is not final:

```
nfa-∅ : Nfa 1
nfa-∅ = record { S = 0F ; δ = λ _ _ → ⟦ 0F ⟧ ; F = ∅ }

nfa-∅-is-empty : (s : String) → ¬ (nfa-∅ ↓ s)
nfa-∅-is-empty (x :: s) r = ⊥-elim (nfa-∅-is-empty s (extractOrL r))
```

The language with just the empty string has two states. The starting state which is final and an error state where all transitions end up:

```
nfa-ϵ : Nfa 2
nfa-ϵ = record { S = 0F ; δ = λ _ _ → ⟦ 1F ⟧ ; F = ⟦ 0F ⟧ }

1F-is-error : (s : String) → ¬ (T (accepts nfa-ϵ 1F s))
```

```
1F-is-error [] z = z
1F-is-error (x :: s) z = ⊥-elim (1F-is-error s (extractOrL z))

nfaϵ-correct : (s : String) → ¬ (s ≡ ϵ) → ¬ (nfa-ϵ ↓ s)
nfaϵ-correct [] a b = a refl
nfaϵ-correct (x :: s) a b = 1F-is-error s (extractOrL b)
```

The language with just a single character string has three states. The construction is defined based on an input character c. The first state is the starting state. The second state is the only final state. The third state is an error state. There is only one transition with character c from the starting state leading to the second state. All other transitions end up in the error state:

```
nfa-c : (c : Σ) → Nfa 3
nfa-c c = record { S = 0F ; δ = δ ; F = ⟦ 1F ⟧ }
  where
    δ : Fin 3 → Σ → Subset 3
    δ 0F k with k ≟ c
    ... | yes p = ⟦ 1F ⟧
    ... | no ¬p = ⟦ 2F ⟧
    δ _ _ = ⟦ 2F ⟧

2F-is-error : ∀{c s} → ¬ (T (accepts (c-nfa c) 2F s ))
2F-is-error {c} {x :: s} d =
  contradiction (extractOrL d) (2F-is-error {c} {s})

nfac-correct : ∀{c}{s} → nfa-c c ↓ s → s ≡ (c :: [])
nfac-correct {c} {x :: []} d with x ≟ c
... | yes p = cong (_:: []) p
nfac-correct {c} {x :: y :: s} d with x ≟ c
... | yes p = contradiction d (2F-is-error {c} {x :: y :: s})
... | no ¬p = contradiction d (2F-is-error {c} {x :: y :: s})
```

3.3.2 Regular expressions to NFAs

We now prove that for every regular expression there is a NFA which recognizes the same language.
For $\langle\rangle$, the equivalent NFA is `nfa-∅`.
For $\langle\epsilon\rangle$, the equivalent NFA is `nfa-ϵ`.
For `Atom c`, we build `nfa-c` passing `c`.
For `(R + S)`, using `union`, we build the union of the two NFAs generated inductively

based on R and S.

For (R · S), using concat, we build the concatenation of the two NFAs generated inductively based on R and S.

For R ⋆, using star, we build the star closure of the NFA generated inductively based on R.

We conclude that a string belongs to the language of a regular expression if and only if the generated NFA accepts it, as a consequence of the correctness proofs nfa-∅-is-empty, nfaϵ-correct, nfac-correct, union-correct, concat-correct, star-correct2 and star-correct1:

```
toNFA : (R : RegExp)
    → ∃₂ λ (n : ℕ) (nfa : Nfa n)
      → ∀ (s : String)
      → s ∈ R ⇔ nfa ↓ s
toNFA ⟨⟩ = 1 , nfa-∅ , λ s → record
    { to = λ ()
    ; from = λ nfa↓s → ⊥-elim (nfa-∅-is-empty s nfa↓s)
    }

toNFA ⟨ϵ⟩ = 2 , nfa-ϵ , iff
  where
  iff : (s : String)
    → s ∈ ⟨ϵ⟩ ⇔ nfa-ϵ ↓ s
  iff [] = record { to = λ _ → tt ; from = λ _ → in-ϵ }
  iff (x ∷ s) = record
    { to = λ ()
    ; from = λ nfa↓xs → ⊥-elim (nfaϵ-correct (x ∷ s) (λ ()) nfa↓xs )
    }

toNFA (Atom c) = 3 , nfa-c c , λ s → to IFF from
  where
  to : ∀{s}
    → s ∈ Atom c
    → nfa-c c ↓ s
  to (in-c c) with c ≟ c
  ... | yes p = tt
  ... | no ¬p = ¬p refl

  from : ∀{s}
    → nfa-c c ↓ s
    → s ∈ Atom c
  from {s} nfa↓s rewrite nfac-correct {c} {s} nfa↓s = in-c c

toNFA (R + F) with toNFA R | toNFA F
... | n , A , w∈R⇔A↓w | m , B , w∈F⇔B↓w =
  suc n ℕ.+ m , union A B , λ s → to s IFF (from s)
  where
  to :  (s : String)
    → s ∈ (R + F)
    → union A B ↓ s
  to s (in+l s∈R)
    = _⇔_.to (union-correct s) (inj₁ (_⇔_.to (w∈R⇔A↓w s) s∈R))
  to s (in+r s∈F)
    = _⇔_.to (union-correct s) (inj₂ (_⇔_.to (w∈F⇔B↓w s) s∈F))

  from : (s : String)
    → union A B ↓ s
    → s ∈ (R + F)
  from s AUB↓s with _⇔_.from (union-correct s) AUB↓s
  ...| inj₁ A↓s = in+l (_⇔_.from (w∈R⇔A↓w s) A↓s)
  ...| inj₂ B↓s = in+r (_⇔_.from (w∈F⇔B↓w s) B↓s)

toNFA (R · F) with toNFA R | toNFA F
... | n , A , w∈R⇔A↓w | m , B , w∈F⇔B↓w =
  suc n ℕ.+ m , concat A B , λ s → to s IFF (from s)
  where
  to : (s : String)
    → s ∈ (R · F)
    → concat A B ↓ s
  to _ (in-· {u} {v} u∈R v∈F) =
    _⇔_.from (concat-correct (u ++ v))
      (u
      , v
      , refl
      , _⇔_.to (w∈R⇔A↓w u) u∈R
      , _⇔_.to (w∈F⇔B↓w v) v∈F)

  from : (s : String)
    → concat A B ↓ s
    → s ∈ (R · F)
  from s AB↓s with _⇔_.to (concat-correct s) AB↓s
  ... | u , v , s≡uv , A↓u , B↓v rewrite s≡uv =
    in-· (_⇔_.from (w∈R⇔A↓w u) A↓u) (_⇔_.from (w∈F⇔B↓w v) B↓v)

toNFA (R ⋆) with toNFA R
... | n , A , s∈R⇔A↓s =
  suc n , star A , λ s → (to s) IFF (from s)
  where
  to : (s : String)
    → s ∈ (R ⋆)
    → star A ↓ s
  to _ in-⋆1 = tt
  to _ (in-⋆2 {u} {v} u∈R v∈R⋆) =
    star-correct1 {_} {u} {v} (_⇔_.to (s∈R⇔A↓s u) u∈R , (to v v∈R⋆))

  lenv<lenau++v : ∀{u v} → (a : Σ) → length v < length (a ∷ u ++ v)
  lenv<lenau++v {[]} {v} a = ≤-refl
  lenv<lenau++v {u ∷ us} {v} a = ≤-step (lenv<lenau++v u)

  star-from-WF : (s : String)
    → star A ↓ s
    → Acc _<_ (length s)
    → s ∈ (R ⋆)
  star-from-WF [] _ _ = in-⋆1
  star-from-WF (a ∷ s) A⋆↓as (acc go) with star-correct2 a s A⋆↓as
  ... | u , v , as≡uv , A↓au , A⋆↓v rewrite as≡uv =
    in-⋆2 (_⇔_.from (s∈R⇔A↓s (a ∷ u)) A↓au)
        (star-from-WF v A⋆↓v (go (length v) (lenv<lenau++v a)))

  from : (s : String)
    → star A ↓ s
    → s ∈ (R ⋆)
  from s A⋆↓s = star-from-WF s A⋆↓s (<-wellFounded (length s))
```

3.3.3 Decidable

The previous result is not just a theorem. It produces a perfectly well-defined NFA which accepts the same language of a given regular expression. We can use it to decide whether or not a string v belongs to $L(F)$.

Given a regular expression, we generate the equivalent NFA, compute the result and return to the regular expressions membership relation with the "⇔" roperty:

```
_∈?_ : (v : String) → (F : RegExp) → Dec (v ∈ F)
v ∈? F with toNFA F
... | _ , A , v∈F⇔A↓v with A ↓? v
... | yes A↓v = yes (_⇔_.from (v∈F⇔A↓v v) A↓v)
... | no ¬A↓v =
  no λ v∈F → contradiction (_⇔_.to (v∈F⇔A↓v v) v∈F) ¬A↓v
```

For example, by evaluating the expression aabaa ∈? a⋆b?a⋆, Agda provides the following membership proof:

```
yes
  (in-·
     (in-· (in-⋆2 (in-c a) (in-⋆2 (in-c a) in-⋆1))
       (in+l (in-c b)))
     (in-⋆2 (in-c a)
       (in-⋆2 (in-c a) (in-⋆2 (in-c a) in-⋆1))))
```

This approach does the same job as the derivatives one previously described. However, in this case, the generated NFA can be reused multiple times for the same regular expressions. Furthermore, it can be transformed into a deterministic automaton that would have a linear time complexity match.

3.4 Pumping Lemma and a Non-regular Langauge

In this section, we present the Pumping Lemma, a property that holds for any finite state automaton. It can be used to prove that a language is not regular. In fact, we present a simple context-free language which is not recognizable by any finite state automaton. We go back to determinism and use DFAs instead of NFAs to simplify the proofs, but as mentioned, these computation models are equivalent and so they both respect this property.

3.4.1 Path

As hinted, we will use the pigeonhole principle and we will apply it to the path of visited states. Given a string s and a DFA A with m states, we define a function which computes the path of visited states during the execution of A with s as input. The path is a vector of Fin m elements of size equal to the length of s. We only consider past states, so when the string is empty, the list of past states is empty.

```
path : ∀{m} → Dfa m
    → Fin m
    → (s : String)
    → Vec (Fin m) (length s)
path A q [] = []
path A q (c :: s) = q :: (path A (δ A q c) s)
```

We can prove a simple lemma which states that the state at an index `i` in the path is equal to the state computed by the extended transition function executed with a prefix of length `i` of the string:

```
lemmaPath : ∀{m}
  → (dfa : Dfa m)
  → (s : String)
  → (i : Fin (length s))
  → (q : Fin m)
  → path dfa q s ! i ≡ δ^ dfa q (take (toℕ i) s)
lemmaPath dfa (c :: s) fzero q = refl
lemmaPath dfa (c :: s) (fsuc i) q = lemmaPath dfa s i (δ dfa q c)
```

3.4.2 Power

A string elevated to the power of `n` is its self-concatenation `n` times. We define it recursively based on the exponent:

```
_^_ : String → ℕ → String
s ^ zero = []
s ^ (suc n) = s ++ s ^ n
```

For example, the string `abc` elevated to the power of `2` generates `abcabc`.
The power operation has some interesting properties, similar to the ones about natural numbers.
For example, $s^n s^m$ is equal to s^{n+m}:

```
^-join-+ : (s : String) (n m : ℕ)
  → (s ^ n) ++ (s ^ m) ≡ s ^ (n + m)
^-join-+ s 0F     0F = refl
^-join-+ s 0F     (suc m) = refl
^-join-+ s (suc n) m rewrite
      ++-assoc s (s ^ n) (s ^ m)
    | sym (^-join-+ s n m) = refl
```

Another example is that $(s^n)^m$ is equal to s^{n*m}:

```
^-join-* : (s : String) (n m : ℕ)
  → (s ^ n) ^ m ≡ s ^ (n * m)
^-join-* s 0F 0F = refl
^-join-* s 0F (suc m) = ^-join-* s 0F m
^-join-* s (suc n) 0F = ^-join-* s n 0F
^-join-* s (suc n) (suc m) rewrite
              ^-join-* s (suc n) m
            | ++-assoc s (s ^ n) (s ^ (m + n * m))
            | ^-join-+ s n (m + n * m)
            | *-comm n (suc m)
            | *-comm m n
            | sym (+-assoc n m (n * m))
            | sym (+-assoc m n (n * m))
            | +-comm m n = refl
```

3.4.3 Pigeonhole principle

The pigeonhole principle states that given two sets A, B and a function $f : A \to B$, if B has less elements than A then there are at least two distinct elements i, j in A such that $f(i) \equiv f(j)$.
In Agda, finite sets can be easily described using the `Fin n` type, and so the pigeonhole proof makes use of this type. The proof can be found in the standard library of Agda. Here is its statement:

```
pigeonhole : ∀ {m n} → m < n → (f : Fin n → Fin m) →
             ∃₂ λ i j → i ≢ j × f i ≡ f j
```

We apply the pigeonhole principle to vectors. Given a vector of size `m` containing elements of type `Fin n`, if `n < m` then there are at least two distinct indexes which point to equal elements. We obtain the following property:

```
pigeonholeVec : ∀{n m}
  → (vec : Vec (Fin n) m)
  → n < m
  → ∃[ i ] ∃[ j ] (
        i <ᶠ j
      × toℕ j ≤ n
      × vec ! i ≡ vec ! j)
```

We will use these indexes to divide a given string into three substrings. Here we define a lemma about this operation. Given a string s and two numbers i, j such that $i < j \leq |s|$, we can divide s in three substrings x, y, z which respect the following constraints:

- x is the prefix of s of length i
- y is not empty
- xy is the prefix of s of length j

Here is the statement:

```
tripartition : (s : String)
  → (i j : Fin (length s))
  → i <ᶠ j
  → ∃[ x ] ∃[ y ] ∃[ z ] (
        s ≡ x ++ y ++ z
      × y ≢ []
      × x ≡ take (toℕ i) s
      × (x ++ y) ≡ take (toℕ j) s
      × length (x ++ y) ≡ toℕ j
    )
```

3.4.4 Returning to same state

When a DFA starts at q and returns to q after running a string s, then it always returns to the same state for any power of s. We show that if $\delta\hat{}(q, s) \equiv q$, then for any number m, we have that $\delta\hat{}(q, s^m) \equiv q$:

```
returns-back : ∀{n}
  → (dfa : Dfa n)
  → (s : String)
  → (q : Fin n)
  → q ≡ δ^ dfa q s
  → ∀(m : ℕ) → q ≡ δ^ dfa q (s ^ m)
returns-back dfa s q eq zero = refl
returns-back dfa s q eq (suc m) with
  returns-back dfa s q eq m | lemma-δ^ dfa s (s ^ m) q
... | ind | lm2 rewrite sym eq = trans ind (sym lm2)
```

We extend `returns-back` lemma by adding a prefix and a suffix and beginning at the starting state:

```
pumping-same-state : ∀{n} {dfa : Dfa n}
  → (s : String)
  → (t : String)
  → (u : String)
  → let p = δ^ dfa (S dfa) s in
     p ≡ δ^ dfa p t
  → dfa ↓ (s ++ t ++ u)
  → ∀ (m : ℕ) → dfa ↓ (s ++ t ^ m ++ u)
pumping-same-state {n} {dfa} s t u p≡δ^pt dfa↓stu m with
  returns-back dfa t (δ^ dfa (S dfa) s) p≡δ^pt m
... | pump with   lemma-δ^ dfa (s ++ t)       u       (S dfa)
                | lemma-δ^ dfa s              (t ^ m) (S dfa)
                | lemma-δ^ dfa s              t       (S dfa)
                | lemma-δ^ dfa (s ++ (t ^ m)) u       (S dfa)
... | d1 | d2 | d3 | d4 rewrite
                  trans pump (sym d2)
                | sym (trans p≡δ^pt (sym d3))
                | ++-assoc s t u
                | ++-assoc s (t ^ m) u
                | trans d1 (sym d4) = dfa↓stu
```

3.4.5 Pumping Lemma

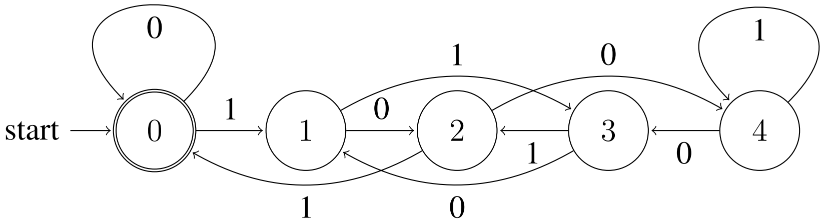
Given a DFA A with m states, there exists a number n such that, for every string $w \in L(A)$, if $|w| > n$ then w can be divided into three substrings x, y, z such that:

- $w \equiv xyz$
- $y \not\equiv \epsilon$
- $|xy| \leq n$
- $\forall k : xy^k z \in L(A)$

One n for which we can be sure this property holds is simply m, the number of states of the DFA.
We use the pigeonholeVec principle applied to the path. Then we use the computed indexes to divide the string with tripartition which generates the strings x, y and z such that $y \not\equiv \epsilon$ and $|xy| \leq m$.
The pigeonholeVec also tells us that there are two repeated states in two different positions in the path which correspond exactly at the end of the computation of x and the end of the computation of y. In fact, with lemmaPath we obtain that: $\delta\hat{}(S, x) \equiv q$ and $\delta\hat{}(q, y) \equiv q$.
Since $\delta\hat{}(q, z) \in F$, using pumping-same-state we conclude that $\forall k : xy^k z \in L(A)$:

```
pumpingLemma : {m : ℕ}
  → (dfa : Dfa m)
  → ∃[ n ] (
    ∀(w : String)
    → dfa ↓ w
    → n < length w
    → ∃[ x ] ∃[ y ] ∃[ z ] (
        w ≡ x ++ y ++ z
        × y ≢ ϵ
        × length (x ++ y) ≤ n
        × ∀(k : ℕ) → dfa ↓ (x ++ y ^ k ++ z)
      )
  )
pumpingLemma {m} dfa = m , base
  where
  base : ∀(w : String)
    → dfa ↓ w
    → m < length w
    → ∃[ x ] ∃[ y ] ∃[ z ] (
        w ≡ x ++ y ++ z
        × y ≢ []
        × length (x ++ y) ≤ m
        × ∀(k : ℕ) → dfa ↓ (x ++ y ^ k ++ z)
      )
  base w dfa↓w m<len_w with
        pigeonholeVec (path dfa (S dfa) w) m<len_w
  ... | i , j , i<j , j≤n , eq  with
        tripartition w i j i<j
  ... | x , y , z , w≡xyz , y≢ϵ , x≡takeI , xy≡takeJ , len_xy≡J with
        lemmaPath dfa w i (S dfa) | lemmaPath dfa w j (S dfa)
  ... | lp1 | lp2 rewrite
        sym x≡takeI | sym xy≡takeJ | sym eq | sym len_xy≡J | w≡xyz =
    x , y , z , ( refl , y≢ϵ , j≤n
        , pumping-same-state x y z
            (trans (sym lp1) (trans lp2 (lemma-δ^ dfa x y (S dfa))))
            dfa↓w
      )
```

For example, we can define a DFA accepting binary strings representing multiples of 5 and then find a substring that can be repeated indefinitely and still preserve the multiple property. The DFA has 6 states where 0 is the starting state and 5 is the error state:



```
dfa-binary-multiples-5 = make-dfa 6 0F 5F (0F :: []) (
        (0F , 0 , 0F)
      :: (0F , 1 , 1F)
      :: (1F , 0 , 2F)
      :: (1F , 1 , 3F)
      :: (2F , 1 , 0F)
      :: (2F , 0 , 4F)
      :: (3F , 1 , 2F)
      :: (3F , 0 , 1F)
      :: (4F , 1 , 4F)
      :: (4F , 0 , 3F)
      :: []
    )
```

When we apply the pumping lemma on it with the string 1110011, which is equal to 115 base-ten, we get that $x = 11$, $y = 100$ and $z = 11$. If we remove y with the power 0, we get that 1111 is accepted, as expected since 1111 is equal to 15 base-ten. If we use for example the power 3, we get the number 1110010010011, which is a multiple of 5 as it is equal to 7315 base-ten:

```
-- [115] dec = [1110011] bin

pumpLem = proj₂
            (pumpingLemma dfa-binary-multiples-5)
            (1 :: 1 :: 1 :: 0 :: 0 :: 1 :: 1 :: [])
            tt
            (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s (s≤s z≤n)))))))

x : String
x = 1 :: 1 :: []

y : String
y = 1 :: 0 :: 0 :: []
```

```
z : String
z = 1 :: 1 :: []

acc1 : dfa-binary-multiples-5 ↓ (x ++ y ^ 0 ++ z)
acc1 = tt

acc2 : dfa-binary-multiples-5 ↓ (x ++ y ^ 3 ++ z)
acc2 = tt
```

3.4.6 A Non-regular language

A language not recognizable by any DFA is called Non-regular language. For example, the context-free language $L = \{1^n 0^n | \, n \geq 0\}$ is not regular. We show that the assumption that there is a DFA that recognizes L is absurd.

Here is the definition of the language:

```
I = '1' :: []
O = '0' :: []

L : ℕ → String
L n = (I ^ n) ++ (O ^ n)

_∈L : String → Set
x ∈L = ∃[ n ] (x ≡ L n)
```

We say that a string x belongs to the language L if there exists a number n such that $x \equiv 1^n 0^n$. We define the following lemmas:

```
exponents-equal : ∀{n m}
  → (I ^ n ++ O ^ m) ∈L
  → n ≡ m

char-^-length-++ : (c : Σ) (n : ℕ) (t : String)
  → length ((c :: []) ^ n ++ t) ≡ n + length t

char-^-length : (c : Σ) (n : ℕ)
  → length ((c :: []) ^ n) ≡ n

xyz-to-power : ∀{n x y z}
  → (I ^ n) ++ (O ^ n) ≡ x ++ y ++ z
  → length (x ++ y) ≤ n
  → y ≢ ϵ
  → ∃[ l ] ∃[ p ] ∃[ q ] (
       x ≡ I ^ l
     × y ≡ I ^ p
     × 0 < p
     × z ≡ (I ^ q) ++ (O ^ n))

absurd-sum : (l p n q : ℕ)
  → 1 ≤ p
  → l + p * (1 + n) + q ≢ n
```

Assume there is a DFA D with m states with the same language of L.
The string 1^m0^m belongs to L and so by assumption, it is also accepted by D. Its length is exactly two times the number of states of D, so we satisfy the condition $m < |1^m0^m|$.
Using the pumping lemma on D and 1^m0^m, we know that there are x, y, z, such that $1^m0^m \equiv xyz$, y is not the empty string, $|xy| \leq m$, and $\forall k : xy^kz \in L(A)$.
The lemma xyz-to-power says that since $1^m0^m \equiv xyz$ and $|xy| \leq m$ and y is not empty, then there are l, $p > 0$, q, such that $x = 1^l$, $y = 1^p$, $z = 1^q0^m$.
We pump y with $k = 1+m$ and get $1^l(1^p)^{1+m}1^q0^m$, which can be simplified to $1^{l+p*(1+m)+q}0^m$ using the power lemmas. Now, since this string is accepted by D, by assumption it also belongs to L.
Our exponents-equal lemma states that if a string of the form 1^a0^b belongs to L, then a must be equal to b. We apply this lemma on the string $1^{l+p*(1+m)+q}0^m$ and obtain $l + p * (1 + m) + q \equiv m$.
We know that $p > 0$ and that l and q are natural numbers. By assuming that $l = 0$, $p = 1$ and $q = 0$, which are the lowest possible values (any other values would just increase the inequality), we still conclude that $1 + m \equiv m$, absurd.
And here is the proof in Agda:

```
L-not-regular : ¬ ∃₂ λ (n : ℕ) (dfa : Dfa n)
                    → ∀ (s : String)
                    → s ∈L ⇔ dfa ↓ s

L-not-regular (n , dfa , s∈L⇔dfa↓s) with proj₂
  (pumpingLemma dfa)
  (L n)
  (_⇔_.to (s∈L⇔dfa↓s (L n)) (n , refl))
  (subst (suc n ≤_)
          (sym (L-length=n+n n))
          (lemmaℕ≤ n (dfa-states>0 dfa))
  )
... | x , y , z , eq , neq , lm , pump with xyz-to-power eq lm neq
... | l , p , q , x≡I^l , y≡I^p , 0<p , z≡I^q++I^n rewrite
  y≡I^p | x≡I^l | z≡I^q++I^n with
  _⇔_.from
    (s∈L⇔dfa↓s (I ^ l ++ I ^ p ^ (1 + n) ++ I ^ q ++ O ^ n))
    (pump (1 + n))
... | fst , snd rewrite
      ^-join-* I p (1 + n)
    | sym (++-assoc (I ^ l) (I ^ (p * (1 + n))) (I ^ q ++ O ^ n))
    | ^-join-+ I l (p * (1 + n))
    | sym (++-assoc (I ^ (l + (p * (1 + n)))) (I ^ q) (O ^ n))
    | ^-join-+ I (l + p * (1 + n)) q
  = absurd-sum l p n q 0<p (exponents-equal (fst , snd))
```

Chapter 4

Conclusion

We formalized regular expressions and NFAs and proved the correctness of two different approaches for solving the matching problem. We showed that finite-state automata are closed under union, concatenation, and star operations and that the regular expressions language class is a subset of the NFAs language class. One further development could be to prove the inverse transformation. Our development ended with the formalization of the pumping lemma, which we used to prove the limits of these computation models. There is also a pumping lemma for the immediate superset of regular languages, context-free languages. The formalization of this class of languages could also be a future extension of the project.

An important conclusion that can be drawn from all this work is that sophisticated type-systems can be used as powerful tools that enforce writing correct code, to such an extent that compilers can be turned into theorem provers. Agda is a clear example of this. On a proof assistant level, the constructive evidence approach used in Agda helps understand the flow of proofs flawlessly. In some cases, though, this also requires to provide constructions for many other seemingly obvious properties, which other theorem provers solve automatically. However, the Agda standard library is immense and it exposes plenty of well-known principles and properties. For instance, the file `Data.Nat.Properties` contains many properties about natural numbers and counts `2000+` lines of proofs.
When firstly approaching Agda, the learning curve can be very steep, but with a solid base in functional programming and Haskell, it becomes straightforward.

Bibliography

[1] Wadler, Philip and Wen Kokke. Programming Language Foundations in Agda. Available at `http://plfa.inf.ed.ac.uk`. 2019.

[2] Agda Wiki. Chalmers and Gothenburg University, 2.4 edn. (2014), `http://wiki.portal.chalmers.se/agda`

[3] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2013). Introduction to Automata Theory, Languages, and Computation (3rd ed.).

[4] Janusz A. Brzozowski, Derivatives of regular expressions, JOURNAL OF THE ACM, 1964, 11, 481–494