# Chapter 4

# Finite Automata

Another way of characterizing regular languages are finite automata (FA)[18]. We will show that the languages of finite automata are exactly the regular languages. Furthermore, we will also derive a decision procedure for equivalence of regular expressions.

## 4.1 Definition

A finite automaton consists of

1. finite set of states $Q$,
2. a starting state $s \in Q$,
3. a set of final states $F \subseteq Q$
4. and a state-transition relation $\delta$.

We define a **run** of a word $w \in \Sigma^*$ on an automaton $A = (Q, s, F, \delta)$ as a sequence of states $\sigma$ such that for every two consequetive positions $i, i+1$ in $\sigma$ there is $(\sigma_i, w_i, \sigma_{i+1}) \in \delta$. A word $w$ is **accepted** by $A$ in state $x$ if and only if there exists a run $\sigma$ of $w$ on $A$ such that $\sigma_0 = x \wedge \sigma_{|\sigma|-1} \in F$. The resulting set of accepted words is denoted by $\mathcal{L}_x(A)$. The **language** of $A$ is exactly $\mathcal{L}_s(A)$ and is denoted $\mathcal{L}(A)$.

### 4.1.1 Non-Deterministic Finite Automata

Finite automata can be **non-deterministic** (NFA) in the sense that there exist multiple distinct runs for a word. This is the case if and only if $\delta$ is not functional.

Listing 4.1: Non-Deterministic Finite Automata

```
Record nfa : Type :=
  { nfa_state :> finType;
```

```
    nfa_s : nfa_state ;
    nfa_fin : pred nfa_state ;
    nfa_step : nfa_state -> char -> pred nfa_state }.
Fixpoint nfa_accept (x: A) w :=
match w with
  | [::] => nfa_fin A x
  | a::w => [ exists y, (nfa_step A x a y) && nfa_accept y w ]
end.
Definition nfa_lang := [pred w | nfa_accept (nfa_s A) w].
```

The acceptance criterion given here avoids the matter of runs. In many cases, this will help us with proofs by in induction on the accepted word. However, we will need runs in some of the proofs. Due to the fact that runs are not unique on NFAs, we give a predicate that decides if a run on $A$ is valid for a word $w$. We then show that the acceptance criterion given above corresponds to the mathematical definition in terms of runs.

```
Fixpoint nfa_run x (xs : seq A) (w: word) {struct xs} :=
match xs,w with
  | y :: xs', a::w' => nfa_step A x a y && nfa_run y xs' w'
  | [::]    , [::]  => true
  | _       , _     => false
end.
Lemma nfa_run_accept x w:
  reflect (exists2 xs, nfa_run x xs w & last x xs \in nfa_fin A)
          (nfa_accept x w).
```

### 4.1.2 Deterministic Finite Automata

For functional $\delta$, we speak of **deterministic** finite automata (DFA). In this case, we write $\delta$ as a function in our Coq development.

Listing 4.2: Deterministic Finite Automata

```
Record dfa : Type :=
  { dfa_state :> finType;
    dfa_s : dfa_state ;
    dfa_fin : pred dfa_state ;
    dfa_step : dfa_state -> char -> dfa_state }.
Fixpoint dfa_accept x w :=
match w with
  | [::] => dfa_fin A x
  | a::w => dfa_accept (dfa_step A x a) w
end.
Definition dfa_lang := [pred w | dfa_accept (dfa_s A) w].
```

Again, we avoid runs in our formalization of the acceptance criterion in favor of a acceptance criterion that is easier to work with in proofs. In this case, however, we can give a function that computes the unique run of a word on $A$. This allows us to give an alternative acceptance criterion that is

closer to the mathematical definition. We also prove that both criteria are equivalent.

```
Fixpoint dfa_run' (x: A) (w: word) : seq A :=
match w with
  | [::] => [::]
  | a::w => (dfa_step A x a) :: dfa_run' (dfa_step A x a) w
end.
Lemma dfa_run_accept x w: last x (dfa_run' x w) \in dfa_fin A = (w \in dfa_accept x).
```

### Equivalence of Automata

**Definition 4.1.1.** *We say that two automata are* ***equivalent*** *if and only if their languages are equal.*

### Equivalence between DFA and NFA

Deterministic and non-deterministic finite automata are equally expressive. One direction is trivial since every DFA can be seen as a NFA. We prove the other direction using the powerset construction. Given NFA $A$, we construct an equivalent DFA $A_{det}$ in the following way:

$$\begin{aligned} Q_{det} &:= 2^Q \\ s_{det} &:= \{s\} \\ F_{det} &:= \{P \mid P \in Q_{det} \wedge P \cap F \neq \emptyset\} \\ \delta_{det} &:= \{(P, a, \bigcup_{p \in P} \{q \mid q \in Q, (p, a, q) \in \delta\}) \mid P \in Q_{det}, a \in \Sigma\}. \\ A_{det} &:= (Q_{det}, s_{det}, F_{det}, \delta_{det}). \end{aligned}$$

The formalization of $A_{det}$ is straight-forward. We leave the set of states {set A} implicit.

```
Definition nfa_to_dfa :=
  {| dfa_s := set1 (nfa_s A);
     dfa_fin := [ pred X: {set A} | [ exists x: A, (x \in X) && nfa_fin A x] ];
     dfa_step := [ fun X a => \bigcup_(x | x \in X) finset (nfa_step A x a) ] |}
.
```

**Lemma 4.1.2.** *For all powerset states $X$ and for all states $x$ with $x \in X$ we have that*

$$\mathcal{L}_x(A) \subseteq \mathcal{L}_X(A_{det}).$$

*Proof.* Let $w \in \mathcal{L}_x(A)$. We proof by induction on $w$ that $w \in \mathcal{L}_X(A_{det})$.

- For $w = \varepsilon$ and $\varepsilon \in \mathcal{L}_x(A)$ we get $x \in F$ from $\varepsilon \in \mathcal{L}_x(A)$. From $x \in X$ we get $X \cap F \neq \emptyset$ and therefore $\varepsilon \in \mathcal{L}_X(A_{det})$.

- For $w = aw'$ and $aw' \in \mathcal{L}_x(A)$ we get $y$ such that $w' \in \mathcal{L}_y(A)$ and $(x, a, y) \in \delta$. The latter gives us $y \in Y$ where $Y$ is such that $(X, a, Y) \in \delta_{det}$. With $y \in Y$ and $w' \in \mathcal{L}_y(A)$ we get which gives us $w' \in \mathcal{L}_Y(A_{det})$ by induction hypothesis. With $(X, a, Y) \in \delta_{det}$ we get $aw' \in \mathcal{L}_X(A_{det})$.

□

**Lemma 4.1.3.** *For all powerset states $X$ and all words $w \in \mathcal{L}_X(A_{det})$ there exists a state $x$ such that*

$$x \in X \wedge w \in \mathcal{L}_x(A).$$

*Proof.* We do an induction on $w \in \Sigma^*$.

- For $w = \varepsilon$ and $\varepsilon \in \mathcal{L}_X(A_{det})$ we get $X \cap F \neq \emptyset$. Therefore, there exists $x$ such that $x \in X$ and $x \in F$. Thus, we have $\varepsilon \in \mathcal{L}_x(A)$.
- For $w = aw'$ and $aw' \in \mathcal{L}_X(A_{det})$ we get $Y$ such that $w' \in \mathcal{L}_Y(A_{det})$ and $(X, a, Y) \in \delta_{det}$. From the induction hypothesis we get $y$ such that $y \in Y$ and $w' \in \mathcal{L}_y(A)$. From $y \in Y$ and $(X, a, Y) \in \delta_{det}$ we get $x$ such that $x \in X$ and $(x, a, y) \in \delta$. Thus, $aw' \in \mathcal{L}_x(A)$.

□

**Theorem 4.1.4.** *The powerset automaton $A_{det}$ accepts the same language as $A$, i.e.*

$$\mathcal{L}(A) = \mathcal{L}(A_{det}).$$

*Proof.* "$\subseteq$" This follows directly from lemma 4.1.2 with $x = s$ and $X = s_{det}$.

"$\supseteq$" From lemma 4.1.3 with $X = s_{det}$ we get $\mathcal{L}_{s_{det}}(A_{det}) \subseteq \mathcal{L}_s(A)$, which proves the claim. □

The formalization of this proof is straight-forward and follows the plan laid out above. The corresponding Lemmas are:

```
Lemma nfa_to_dfa_complete (x: A) w (X: nfa_to_dfa ):
  x \in X -> nfa_accept A x w -> dfa_accept nfa_to_dfa X w.
Lemma nfa_to_dfa_sound (X: nfa_to_dfa) w:
  dfa_accept  nfa_to_dfa  X w -> [ exists x, (x \in X) && nfa_accept A x w ].
Lemma nfa_to_dfa_correct :  nfa_lang  A =i dfa_lang  nfa_to_dfa .
```

## 4.2 Connected Components

Finite automata can have isolated subsets of states that are not reachable from the starting state. These states can not contribute to the language of the automaton since there are no runs from the starting state to any of those unreachable states. It will later be useful to have automata that only contain reachable states. Therefore, we define a procedure to extract the connected component containing the starting state from a given automaton.

**Definition 4.2.1.** *Let* $A = (Q, s, F, \delta)$ *be a DFA. We define* reachable1 *such that for all* $x$ *and* $y$, $(x, y) \in$ reachable1 $\iff \exists a, (x, a, y) \in \delta$. *We define* reachable $:= \{y \mid (s, y) \in \text{reachable1}^*\}$, *where* reachable1$^*$ *denotes the transitive closure of* reachable1. *With this, we can define the connected automaton* $A_c$*:*

$$\begin{aligned} Q_c &:= Q \cap \mathit{reachable} \\ s_c &:= s \\ F_c &:= F \cap \mathit{reachable} \\ \delta_c &:= \{(x, a, y) \mid (x, a, y) \in \delta \wedge x, y \in Q_c\} \\ A_c &:= (Q_c, s_c, F_c, \delta_c). \end{aligned}$$

We make use of SSReflect's *connect* predicate to extract a sequence of all states reachable from $s$. From this, we construct a finite type and use that as the new set of states. These new states carry a proof of reachability. We also have to give a transition function that ensures transitions always end in reachable states.

```
Definition reachable1 := [ fun x y => [ exists a, dfa_step A1 x a == y ] ].
Definition reachable := enum (connect reachable1 (dfa_s A1)).
Lemma reachable0 : dfa_s A1 \in reachable.
Lemma reachable_step x a: x \in reachable -> dfa_step A1 x a \in reachable.
Definition dfa_connected :=
  {| dfa_s := SeqSub reachable0;
     dfa_fin := fun x => match x with SeqSub x _ => dfa_fin A1 x end;
     dfa_step := fun x a => match x with
       | SeqSub _ Hx => SeqSub (reachable_step _ a Hx)
       end |}.
```

Make this understandable

**Lemma 4.2.2.** *For every state* $x \in$ reachable *we have that*

$$\mathcal{L}_x(A_c) = \mathcal{L}_x(A).$$

*Proof.* "$\subseteq$" Trivial. "$\supseteq$" We do an induction on $w$.

- For $w = \varepsilon$ we have $\varepsilon \in \mathcal{L}_x(A)$ and therefore $x \in F$. With $x \in \mathit{reachable}$ we get $x \in F_c$. Thus, $\varepsilon \in \mathcal{L}_x(A_c)$.
- For $w = aw'$ we have have $y \in Q$ such that $(x, a, y) \in \delta$ and $w' \in \mathcal{L}_y(A)$. From $x \in$ reachable we get $y \in$ reachable by transitivity. Therefore, $(x, a, y) \in \delta_c$. The induction hypothesis gives us $w' \in \mathcal{L}_y(A_c)$. Thus, $aw' \in \mathcal{L}_x(A_c)$.

□

**Theorem 4.2.3.** *The language of the connected automaton $A_c$ is identical to that of the original automaton $A$, i.e.*

$$\mathcal{L}(A) = \mathcal{L}(A_c).$$

*Proof.* By reflexivity, we have $s \in$ reachable. We use lemma 4.2.2 with $x = s$ to prove the claim. □

The formalization of lemma 4.2.2 and theorem 4.2.3 is straight-forward.

```
Lemma dfa_connected_correct' x (Hx: x \in reachable) :
  dfa_accept dfa_connected (SeqSub Hx) =i dfa_accept A1 x.
Lemma dfa_connected_correct: dfa_lang dfa_connected =i dfa_lang A1.
```

To make use of the fact that $A_c$ is fully connected, we will proof a characteristic property of $A_c$.

**Definition 4.2.4.** *A **representative** of a state $x$ is a word $w$ such that the unique run of $w$ on $A_c$ ends in $x$.*

**Lemma 4.2.5.** *We can give a representative for every state $x \in Q_c$.*

*Proof.* $x$ carries a proof of reachability. From this, we get a path through the graph of reachable1 that ends ind $x$. We build the representative by extracting the edges of the path and building a word from those. □

Choice?

The formalization of theorem 4.2.5 includes a more general version of the theorem, which facilitates the proof by induction over the path.

```
Lemma dfa_connected_repr' (x y: dfa_connected):
  connect reachable1_connected y x ->
  exists w, last y (dfa_run' dfa_connected y w) = x.
Lemma dfa_connected_repr x :
  exists w, last (dfa_s dfa_connected) (dfa_run dfa_connected w) = x.
```

## 4.3 Emptiness

Given an automaton $A$, we can check if $\mathcal{L}(A) = \emptyset$. We simply obtain the connected automaton of $A$ and check if there are any final states left.

**Theorem 4.3.1.** *The language of the connected automaton $A_c$ is empty if and only if its set of final states $F_c$ is empty, i.e.*

$$\mathcal{L}(A) = \emptyset \iff F_c = \emptyset.$$

*Proof.* By theorem 4.2.3 we have $\mathcal{L}(A) = \mathcal{L}(A_c)$. Therefore, it suffices to show

$$\mathcal{L}(A_c) = \emptyset \iff F_c = \emptyset.$$

"⇐" We have $\mathcal{L}(A_c) = \emptyset$ and have to show that for all $x \in Q_c$, $x \notin F_c$. Let $x \in Q_c$. By lemma 4.2.5 we get $w$ such that the unique run of $w$ on $A_c$ ends in $x$. We use $\mathcal{L}(A_c) = \emptyset$ to get $w \notin \mathcal{L}(A_c)$, which implies that the run of $w$ on $A_c$ ends in a non-final state. By substituting the last state of the run by $x$ we get $x \notin F_c$. "⇒" We have $F_c = \emptyset$ and have to show that for all words $w$, $w \notin \mathcal{L}(A_c)$. We use $F_c = \emptyset$ to show that the last state of the run of $w$ on $A_c$ is non-final. Thus, $w \notin \mathcal{L}(A_c)$.

Thus, emptiness is decidable. □

The formalization of lemma 4.3.1 is split in two parts to facilitate its application.

```
Definition dfa_lang_empty := #|dfa_fin  dfa_connected|  == 0.
Lemma dfa_lang_empty_complete: dfa_lang dfa_connected =i pred0 -> dfa_lang_empty.
Lemma dfa_lang_empty_sound: dfa_lang_empty -> dfa_lang dfa_connected =i pred0.
Lemma dfa_lang_empty_correct:
    reflect  (dfa_lang  A1 =i pred0)
             dfa_lang_empty.
```

## 4.4 Deciding Equivalence of Finite Automata

Given finite automata $A_1$ and $A_2$, we construct DFA $A$ such that the language of $A$ is the symmetric difference of the languages of $A_1$ and $A_2$, i.e.,

$$\mathcal{L}(A) := \mathcal{L}(A_1) \ominus \mathcal{L}(A_2) = \mathcal{L}(A_1) \cap \neg\mathcal{L}(A_2) \cup \mathcal{L}(A_2) \cap \neg\mathcal{L}(A_1).$$

**Theorem 4.4.1.** *The equivalence of $A_1$ and $A_2$ is decidable, i.e.*

$$\mathcal{L}(A_1) = \mathcal{L}(A_2) \text{ if and only if } \mathcal{L}(A) \text{ is empty.}$$

*Proof.* The correctness of this procedure follows from the properties of the symmetric difference operator, i.e.

$$\mathcal{L}(A_1) \ominus \mathcal{L}(A_2) = \emptyset \Leftrightarrow \mathcal{L}(A_1) = \mathcal{L}(A_2).$$

Thus, equivalence is decidable. □

Listing 4.3: Formalization of theorem 4.4.1

```
Definition  dfa_sym_diff  :=
    dfa_disj  (dfa_conj  A1 (dfa_compl A2)) (dfa_conj  A2 (dfa_compl A1)).
Lemma dfa_sym_diff_correct:
    dfa_lang_empty  dfa_sym_diff  <-> dfa_lang A1 =i dfa_lang A2.
```

## 4.5 Regular Expressions and Finite Automata

We prove that there is a finite automaton for every extended regular expression and vice versa. In fact, we can give a standard regular expression for every finite automaton. With this, we will prove that extended regular expressions are equivalent to standard regular expressions, thereby proving closure under intersection and negation.

### 4.5.1 Regular Expressions to Finite Automata

We prove that there exists an equivalent automaton for every extended regular expression. The structure of this proof is given by the inductive nature of regular expressions. For every constructor, we provide an equivalent automaton.

Depending on the constructor of the regular expression, we will construct an equivalent DFA or NFA. Void, Eps, Dot, Atom, Plus, And and Not are very easy to implement on DFAs, whereas Star and Conc lend themselves well to NFAs.

#### Void

**Definition 4.5.1.** *We define an empty DFA with a single, non-accepting state, i.e.*

$$A_\emptyset := (\{t\}, t, \emptyset, \{(t, a, t) \mid a \in \Sigma\}).$$

**Lemma 4.5.2.** *The language of the empty DFA is empty, i.e.*

$$\mathcal{L}(E) = \emptyset.$$

*Proof.* $A_\emptyset$ has no final states, i.e. no run can end in a final state. □

```
Definition dfa_void :=
  {| dfa_s := tt;
     dfa_fin := pred0;
     dfa_step := [fun x a => tt] |}.
Lemma dfa_void_correct x w: ~~ dfa_accept dfa_void x w.
```

#### Eps

**Definition 4.5.3.** *We define an automaton that accepts only the empty word, i.e.*

$$A_\varepsilon := (\{t, f\}, t, \{f\}, \{(x, a, f) \mid x \in \{t, f\}, a \in \Sigma\}).$$

**Lemma 4.5.4.** *$A_\varepsilon$ accepts no word in state $f$, i.e. for all $w$,*

$$w \notin \mathcal{L}_f(A_\varepsilon).$$

*Proof.* Let $w \in \Sigma^*$. We do an induction on $w$. For $w = \varepsilon$ we get $\varepsilon \notin \mathcal{L}_f(A_\varepsilon)$ by $f \notin F_\varepsilon$. For $w = aw'$ we have $w' \notin \mathcal{L}_f(A_\varepsilon)$. Furthermore, $(f, a, f) \in \delta_\varepsilon$. Therefore, $aw' \notin \mathcal{L}_f(A_\varepsilon)$. □

**Lemma 4.5.5.** *The language of $A_\varepsilon$ is exactly the singleton set containing the empty word, i.e.*

$$\mathcal{L}(A_\varepsilon) = \{\varepsilon\}.$$

*Proof.* Let $w \in \Sigma^*$. We do an induction on $w$. For $w = \varepsilon$ we have $\varepsilon \in \mathcal{L}(A_\varepsilon)$ and $\varepsilon \in \{\varepsilon\}$. Therefore, both directions are trivial. For $w = aw'$ we consider both directions independently.
"⇒" We have $(t, a, f) \in \delta_\varepsilon$ and $w' \in \mathcal{L}_f(A_\varepsilon)$. By lemma 4.5.4, this is a contradiction.
"⇐" We get a straigt-forward contradiction from $aw' \in \{\varepsilon\}$. □

```
Definition dfa_eps :=
  {| dfa_s := true;
     dfa_fin := pred1 true;
     dfa_step := [fun x a => false] |}.
Lemma dfa_eps_correct: dfa_lang dfa_eps =i pred1 [::].
```

reflect?

### Dot

**Definition 4.5.6.** *We define an automaton that accepts the set of all singleton words, i.e.*

$$A_{Dot} := (\{s, t, f\}, s, \{t\}, \{(s, a, t) \mid a \in \Sigma\} \cup \{(x, a, f) \mid x \in \{t, f\}, a \in \Sigma\}).$$

**Lemma 4.5.7.** *$A_{Dot}$ does not accept any word in state $f$, i.e. $\mathcal{L}_f(A_{Dot}) = \emptyset$.*

*Proof.* We prove this by induction on $w \in \Sigma^*$. For $w = \varepsilon$ we have $\varepsilon \notin \mathcal{L}_f(A_{Dot})$ by $f \notin F_{Dot}$. For $w = aw'$ we have $(f, a, f) \in \delta_{Dot}$ and $w' \notin \mathcal{L}_f(A_{Dot})$ by induction hypothesis. Thus, $aw' \notin \mathcal{L}_f(A_{Dot})$. □

**Lemma 4.5.8.** *$A_{Dot}$ accepts exactly the empty word in state $t$, i.e. $\mathcal{L}_t(A_{Dot}) = \{\varepsilon\}$.*

*Proof.* Let $w \in \Sigma^*$. We do a case distinction on $w$. For $w = \varepsilon$ we have $\varepsilon in \mathcal{L}_t(A_{Dot})$ by $t \in F_{Dot}$. We also have $\varepsilon \in \{\varepsilon\}$. For $w = aw'$ we get $(t, a, f) \in \delta_{Dot}$. Since $aw' \notin \{\varepsilon\}$ it suffices to show that $w' \notin \mathcal{L}_f(A_{Dot})$, which we have by lemma 4.5.7. □

```
Definition dfa_dot :=
  {| dfa_s := None;
     dfa_fin := pred1 (Some true);
     dfa_step := [fun x b => if x == None then Some true else Some false ] |}.
```

**Lemma** dfa_dot_correct'' w: ∼∼ dfa_accept dfa_dot (Some false) w.
**Lemma** dfa_dot_correct' w: dfa_accept dfa_dot (Some true) w = (w == [::]).
**Lemma** dfa_dot_correct w: dfa_lang dfa_dot w = (size w == 1).

reflect?

### Not

**Definition 4.5.9.** *Given DFA $A = (Q, s, F, \delta)$, the complement automaton $A_\neg$ is constructed by switching accepting and non-accepting states, i.e.*

$$A_\neg := (Q, s, Q \backslash F, \delta).$$

**Lemma 4.5.10.** *For every state $x \in Q$, we have that $w \in \Sigma^*$ is accepted in $x$ by $A_\neg$ if and only if it is not accepted in $x$ by $A$, i.e. $\mathcal{L}_x(A_\neg) = \Sigma^* \backslash \mathcal{L}_x(A)$*

*Proof.* We do an induction on $w$. For $w = \varepsilon$ we have $\varepsilon \in \mathcal{L}_x(A_\neg) \iff \varepsilon \in \mathcal{L}_x(A)$ from $x \in F \iff x \notin Q \backslash F$. For $w = aw'$ we get $(y, a, x) \in \delta$. By induction hypothesis, $w' \in \mathcal{L}_x(A_\neg) \iff w' \notin \mathcal{L}_x(A)$ Thus, $aw' \in \mathcal{L}_y(A_\neg) \iff aw' \notin \mathcal{L}_y(A)$. □

**Lemma 4.5.11.** *$A_\neg$ accepts the complement language of $A$, i.e. $\mathcal{L}(A_\neg) = \Sigma^* \backslash \mathcal{L}(A)$.*

*Proof.* This follows directly from lemma 4.5.10 with $x = s$. □

### Plus

**Definition 4.5.12.** *Given DFAs $A_1 = (Q_1, s_1, F_1, \delta_1)$ and $A_2 = (Q_2, s_2, F_2, \delta_2)$ we construct the disjunction automaton in the following way:*

$$\begin{aligned}
Q_\vee &:= Q_1 \times Q_2 \\
s_\vee &:= (s_1, s_2) \\
F_\vee &:= \{(x_1, x_2) \mid x_1 \in F_1 \vee x_2 \in F_2\} \\
\delta_\vee &:= \{((x_1, x_2), a, (y_1, y_2)) \mid a \in \Sigma, (x_1, a, y_1) \in \delta_1, (x_2, a, y_2) \in \delta_2\} \\
A_\vee &:= (Q_\vee, s_\vee, F_\vee, \delta_\vee).
\end{aligned}$$

**Lemma 4.5.13.** *For every state $(x_1, x_2) \in Q_\vee$, we have that*

$$\mathcal{L}_{(x_1,x_2)}(A_\vee) = \mathcal{L}_{x_1}(A_1) \cup \mathcal{L}_{x_2}(A_2).$$

*Proof.* We do a proof by induction on $w \in \Sigma^*$. For $w = \varepsilon$ we have, by definition of $F_\vee$, $\varepsilon \in \mathcal{L}_{(x_1,x_2)}(A_\vee) \iff \varepsilon \in \mathcal{L}_{x_1}(A_1) \vee \varepsilon \in \mathcal{L}_{x_2}(A_2)$.
For $w = aw'$ we get $(x_1, a, y_1) \in \delta_1$ and $(x_2, a, y_2) \in \delta_2$. By induction hypothesis, we also have $w' \in \mathcal{L}_{x_1}(A_1) \vee w' \in \mathcal{L}_{x_2}(A_2)$. Thus, we get $aw' \in \mathcal{L}_{y_1}(A_1) \vee aw' \in \mathcal{L}_{x_2}(A_2)$. □

**Lemma 4.5.14.** $\mathcal{L}(A_\vee) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.

*Proof.* This follows directly from lemma 4.5.13 with $x = (s_1, s_2)$. □

```
Definition dfa_disj :=
  {| dfa_s := (dfa_s A1, dfa_s A2);
     dfa_fin := (fun q => let (x1,x2) := q in dfa_fin A1 x1 || dfa_fin A2 x2);
     dfa_step := [fun x a => (dfa_step A1 x.1 a, dfa_step A2 x.2 a)] |}.
Lemma dfa_disj_correct' x:
  [ predU dfa_accept A1 x.1 & dfa_accept A2 x.2 ]
    =i dfa_accept dfa_disj x.
Lemma dfa_disj_correct:
  [ predU dfa_lang A1 & dfa_lang A2 ]
    =i dfa_lang dfa_disj .
```

### And

**Definition 4.5.15.** *Given DFAs* $A_1 = (Q_1, s_1, F_1, \delta_1)$ *and* $A_2 = (Q_2, s_2, F_2, \delta_2)$ *we construct the conjunction automaton in the following way:*

$$\begin{aligned} Q_\wedge &:= Q_1 \times Q_2 \\ s_\wedge &:= (s_1, s_2) \\ F_\wedge &:= \{(x_1, x_2) \mid x_1 \in F_1 \wedge x_2 \in F_2\} \\ \delta_\wedge &:= \{((x_1, x_2), a, (y_1, y_2)) \mid a \in \Sigma, (x_1, a, y_1) \in \delta_1, (x_2, a, y_2) \in \delta_2\} \\ A_\wedge &:= (Q_\wedge, s_\wedge, F_\wedge, \delta_\wedge). \end{aligned}$$

**Lemma 4.5.16.** *For every state* $(x_1, x_2) \in Q_\wedge$*, we have that*

$$\mathcal{L}_{(x_1,x_2)}(A_\wedge) = \mathcal{L}_{x_1}(A_1) \cup \mathcal{L}_{x_2}(A_2).$$

*Proof.* This proof is very similar to lemma 4.5.13. We do a proof by induction on $w \in \Sigma^*$. For $w = \varepsilon$ we have, by definition of $F_\wedge$, $\varepsilon \in \mathcal{L}_{(x_1,x_2)}(A_\wedge) \iff \varepsilon \in \mathcal{L}_{x_1}(A_1) \wedge \varepsilon \in \mathcal{L}_{x_2}(A_2)$.
For $w = aw'$ we get $(x_1, a, y_1) \in \delta_1$ and $(x_2, a, y_2) \in \delta_2$. By induction hypothesis, we also have $w' \in \mathcal{L}_{x_1}(A_1) \wedge w' \in \mathcal{L}_{x_2}(A_2)$. Thus, we get $aw' \in \mathcal{L}_{y_1}(A_1) \wedge aw' \in \mathcal{L}_{x_2}(A_2)$. □

**Lemma 4.5.17.** $\mathcal{L}(A_\wedge) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

*Proof.* This follows directly from lemma 4.5.16 with $x = (s_1, s_2)$. □

```
Definition dfa_conj :=
 {| dfa_s := (dfa_s A1, dfa_s A2);
    dfa_fin := (fun x => dfa_fin A1 x.1 && dfa_fin A2 x.2);
    dfa_step := [fun x a => (dfa_step A1 x.1 a, dfa_step A2 x.2 a)] |}.
```

```
Lemma dfa_conj_correct' x1 x2 :
  [ predI  dfa_accept  A1 x1 & dfa_accept A2 x2 ]
  =i dfa_accept  dfa_conj  (x1,  x2).
Lemma dfa_conj_correct:
  [ predI  dfa_lang  A1 & dfa_lang A2 ]
  =i dfa_lang  dfa_conj .
```

### Conc

**Definition 4.5.18.** *Given two NFAs $A_1 = (Q_1, s_1, F_1, \delta_1)$ and $A_2 = (Q_2, s_2, F_2, \delta_2)$ we construct the concatenation automaton in the following way:*

$$\begin{aligned} Q_{Conc} &:= Q_1 \uplus Q_2 \\ s_{Conc} &:= s_1 \\ F_{Conc} &:= \textbf{TODO} \\ \delta_{Conc} &:= \delta_1 \cup \delta_2 \cup \{(x, a, y) \mid x \in Q_1, y \in Q_2, (s_2, a, y) \in \delta_2\} \\ A_{Conc} &:= (Q_{Conc}, s_{Conc}, F_{Conc}, \delta_{Conc}). \end{aligned}$$

Case dist. on $s_2 \in F_2$

**Lemma 4.5.19.** *Every run of $A_2$ can be mapped to a run in $A_3$.*

symbol for empty run

*Proof.* Let $\sigma$ be a run starting in $x$ for $w \in \Sigma^*$ on $A_2$. We do an induction on $\sigma$. For $|\sigma| = 0$we have $w = \varepsilon$. Therefore, we have that $\sigma$ is also a run starting in $x$ for $\varepsilon$ on $A_{Conc}$ For $\sigma = y\sigma'$ we have $w = aw'$, $(x, a, y) \in \delta_2$. By definition of $\delta_{Conc}$ we also have $(x, a, y) \in \delta_{Conc}$. By induction hypothesis, we have that $\sigma'$ is a run for $w'$ starting in $y$ on $A_{Conc}$. Thus, $y\sigma'$ is a run for $aw'$ starting in $x$ on $A_{Conc}$. □

Include all proofs

### 4.5.2 Deciding Equivalence of Regular Expressions

Based on our procedure to construct an equivalent automaton from a regular expression, we can decide equivalence of regular expressions. Given $r_1$ and $r_2$, we construct equivalent DFA $A_1$ and $A_2$ as above.

### 4.5.3 Finite Automata to Regular Expressions

We prove that there is an equivalent standard regular expression for every finite automaton. There are three ways to prove this.

The first one is a method called "**state removal**" [11] (reformulated in [17]), which works by sequentially building up regular expressions on the edges between states. In every step, one state is removed and its adjacent states' edges are updated to incorporate the missing state into the regular

expression. Finally, only two states remain. The resulting edges can be combined to form a regular expression that recognizes the language of the initial automaton.

The second method is known as "**Brzozowski's method**" [12] and builds upon Brzowoski derivates of regular expressions. This method is algebraic in nature and arrivates at a regular expression by solving a system of linear equations on regular expressions. Every state is assigned an unkown regular expression. The intuition of these unkown regular expressions is that they recognize the words accepted in their associated state. The system is solved by substituition and Arden's lemma [3]. The regular expressions associated with the starting state recognizes the language of the automaton.

The third method, which we chose for our development, is due to Kleene [22]. It is known as the "**transitive closure method**". This method recursively constructs a regular expression that is equivalent to the given automaton. For the remainder of the chapter, we assume that we are given a DFA $A = (Q, s, F, \delta)$.

The idea of the transitive closure method is that we can give a regular expression to describe the path between any two states $x$ and $y$. This regular expression accepts all words whose run $\sigma$ on $A$ starting in $x$ ends in $y$. In fact, we can even give such a regular expression if we limit the set of paths through which the run is allowed to pass. We will call this set $X$. Passing through, here, means that the restriction applies only to states that are traversed, i.e. not to the beginning or end of the run.

If we take $X$ to be the empty set, we only consider two types of runs. First, if $x \neq y$, every transition from $x$ to $y$ constitutes one (singelton) word. Conversely, if there is a word which does not pass through a state and whose run on $A$ starts in $x$ and ends in $y$, it can only be a singleton word consisting of one of the transitions from $x$ to $y$. Therefore, the corresponding regular expression is the disjunction of all transitions from $x$ to $y$. These transitions constitutes all possible words that lead from $x$ to $y$ without passing through any state.

Otherweise, if $x = y$, we also have to consider the empty word, since its run on $A$ starts in $x$ and ends in $y$. Thus, the corresponding regular expression is the disjunction of all transitions from $x$ to $y$ *and* $\varepsilon$.

In the case of a non-empty $X$, we make the following observation. If we pick an element $z \in X$, then any run $\sigma$ from $x$ to $y$ either passes through $z$, or does not pass through $z$. If it does, we can split it into three parts.

(i) The first part contains the prefix of $\sigma$ which contains all states up to the first occurrence of $z$ that is not the starting state.

(ii) The second part contains that part of the remainder of $\sigma$ which contains all further occurences of $z$, though not the last state if that is $z$.

(iii) The third part contains the remainder.

Parts (i) and (iii) can easily be expressed in terms of $X\backslash\{z\}$. Part (ii) can be further decomposed into runs from $z$ to $z$ that do not pass through $z$. Thus, part (ii) can also be expressed in terms of $X\backslash\{z\}$ with the help of the $^*$ operator.

If $\sigma$ does not pass through $z$, it is covered by the regular expression for pathes from $x$ to $y$ restricted to $X\backslash\{z\}$.

**Definition 4.5.20.** *Let $X \subseteq Q$. Let $x, y \in Q$. We define $R$ recursively on $|X|$:*

correct?

$$R^X_{x,y} := \begin{cases} \sum\limits_{\substack{a\in\Sigma \\ (x,a,y)\in\delta}} a & \text{if } X = \emptyset \wedge x! = y; \\ \sum\limits_{\substack{a\in\Sigma \\ (x,a,y)\in\delta}} a + \varepsilon & \text{if } X = \emptyset \wedge x = y; \\ R^{X\backslash\{z\}}_{x,z}(R^{X\backslash\{z\}}_{z,z})^* R^{X\backslash\{z\}}_{z,y} + R^{X\backslash\{z\}}_{x,y} & \text{if } \exists z \in X. \end{cases}$$

The formalization of $R$ is more involved than its mathematical definition. We give some auxiliary definitions to keep the definition of $R$ as compact and readable as possible. nPlus is $\sum$ on regular expressions. dfa_step_any is the list of transitions from $x$ to $y$. R0 covers the case of $X = \emptyset$.

explain measure

```
Definition nPlus rs := foldr (@Plus char) (Void _) rs.
Definition dfa_step_any x y := enum ( [pred a | dfa_step A x a == y] ).
Definition R0 x y := let r := nPlus (map (@Atom _) (dfa_step_any x y)) in
      if x == y then Plus r (Eps _) else r.
Function R (X: {set A}) (x y: A) {measure [fun X => #|X|] X} :=
  match [pick z in X] with
  | None => R0 x y
  | Some z => let X' := X :\ z in
     Plus (Conc (R X' x z) (Conc (Star (R X' z z)) (R X' z y))) (R X' x y)
  end.
```

Following the intuition given above, we try to express $\mathcal{L}(A)$ by $R$. Based on the observation that every accepted word has a run from $s$ to some state $f \in F$, we only have to combine the corresponding regular expressions $R^Q_{s,f}$ to form a regular expression for $\mathcal{L}(A)$. The goal of this chapter is to prove the following theorem.

**Theorem 4.5.21.** *$\mathcal{L}(A)$ is recognizable by a regular expression, i.e.*

$$\mathcal{L}(\sum_{f\in F} R^Q_{s,f}) = \mathcal{L}(A).$$

In order to prove this theorem, we will first define a predicate in terms of runs that corresponds to $R^X_{x,y}$. We call this predicate $L^X_{x,y}$ and define it such that it includes those words whose runs on $A$ starting in $x$ only pass through states in $X$ and end in $y$. This is a direct implementation of the idea behind $R^X_{x,y}$. Then, we show that $L^{X\cup\{z\}}$ can be expressed in terms of $L^X$. We prove that we can express $\mathcal{L}(A)$ by $\bigcup_{f\in F} L^Q_{s,f}$. Finally, we show that $R^X_{x,y} = L^X_{x,y}$, thus proving theorem 4.5.21.

**Definition 4.5.22.** *Let $w \in \Sigma^*$. Let $X \subseteq Q$, and $x, y \in X$. Let $\sigma$ be the run of $w$ on $A$ starting in $x$. We define $L^X_{x,y}$ such that*

$$w \in L^X_{x,y} \iff \sigma_{|\sigma|-1} = y \ \wedge\ \forall i \in [1, |\sigma| - 2].\ \sigma_i \in X.$$

The formalization of $L$ requires some infrastructure. To check the second condition of $L$, we want to be able to state properties of all but the last items in a run. We define a function to remove the last element from a sequence.

```
Fixpoint belast (xs: seq X) :=
  match xs with
    | [::]  => [::]
    | [:: x] => [::]
    | x :: xs => x::(belast xs)
  end.
Definition allbutlast xs := all p (belast xs).
Definition L (X: {set A}) (x y: A) :=
    [pred w | (last x (dfa_run' A x w) == y)
              && allbutlast (mem X) (dfa_run' A x w) ].
```