

## 4 Finite Automata

Another way of characterizing regular languages are finite automata. We will show that the languages of finite automata are exactly those recognized by regular expressions. Furthermore, we will also derive a decision procedure for emptiness of an automaton's language. Based on that, we will give a decision procedure for equivalence of regular expressions. Finally, we prove that extended regular expressions and standard regular expressions are equally expressive and, thereby, that extended regular expressions are equivalent to regular languages.

### 4.1 Definition

A finite automaton [18] 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 consecutive positions  $i, i+1$  in  $\sigma$  we have  $(\sigma_i, w_i, \sigma_{i+1}) \in \delta$ . 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 may exist multiple distinct runs for a word.

**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.
```

## 4 Finite Automata

**Definition**  $\text{nfa\_lang} := [\text{pred } w \mid \text{nfa\_accept } (\text{nfa\_s } A) w]$ .

The acceptance criterion given here avoids the matter of runs. In many cases, this will help us with proofs by 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 sequence of states is a run on  $A$  for a word  $w$ . We then show that the acceptance criterion given above corresponds to the mathematical definition in terms of runs.

**Fixpoint**  $\text{nfa\_run } x \text{ (xs : seq A) (w: word) \{struct xs\} :=$   
**match**  $\text{xs}, w$  **with**  
 |  $y :: xs', a :: w' \Rightarrow \text{nfa\_step } A \ x \ a \ y \ \&\& \ \text{nfa\_run } y \ xs' \ w'$   
 |  $[], [] \Rightarrow \text{true}$   
 |  $-, - \Rightarrow \text{false}$   
**end.**

**Lemma**  $\text{nfa\_run\_accept } x \ w:$   
 reflect  $(\text{exists2 } xs, \text{nfa\_run } x \ xs \ w \ \& \ \text{last } x \ xs \ \backslash \text{in } \text{nfa\_fin } A)$   
 $(\text{nfa\_accept } x \ w).$

### 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.

**Record**  $\text{dfa} : \text{Type} :=$   
 {  $\text{dfa\_state} :> \text{finType};$   
 $\text{dfa\_s} : \text{dfa\_state};$   
 $\text{dfa\_fin} : \text{pred } \text{dfa\_state};$   
 $\text{dfa\_step} : \text{dfa\_state} \rightarrow \text{char} \rightarrow \text{dfa\_state} \}$ .

**Fixpoint**  $\text{dfa\_accept } x \ w :=$   
**match**  $w$  **with**  
 |  $[], [] \Rightarrow \text{dfa\_fin } A \ x$   
 |  $a :: w \Rightarrow \text{dfa\_accept } (\text{dfa\_step } A \ x \ a) \ w$   
**end.**

**Definition**  $\text{dfa\_lang} := [\text{pred } w \mid \text{dfa\_accept } (\text{dfa\_s } A) w]$ .

Again, we avoid runs in our formalization of the acceptance criterion in favor of an 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**  $\text{dfa\_run}' (x: A) (w: word) : \text{seq } A :=$   
**match**  $w$  **with**  
 |  $[], [] \Rightarrow []$   
 |  $a :: w \Rightarrow (\text{dfa\_step } A \ x \ a) :: \text{dfa\_run}' (\text{dfa\_step } A \ x \ a) \ w$   
**end.**

**Lemma**  $\text{dfa\_run\_accept } x \ w: \text{last } x \ (\text{dfa\_run}' \ x \ w) \ \backslash \text{in } \text{dfa\_fin } A = (w \ \backslash \text{in } \text{dfa\_accept } x).$

### $\varepsilon$ -Transitions

Non-deterministic finite automata with  $\varepsilon$ -transitions are based on the non-deterministic automata described above. In addition to all possible transitions of a normal NFA, they also allow for transitions that can be taken regardless of the input. These transitions are marked with the  $\varepsilon$  character. They do not “consume” a character of the input and can be taken arbitrarily often. The main disadvantage is that the properties of  $\varepsilon$ -transitions do not warrant a one-to-one correspondence between run length and word length. Specifically, runs may be much longer than their corresponding words. Thus, induction on runs no longer directly translates to induction on the size of the word. To avoid nested inductions, we decided not to include  $\varepsilon$ -transitions in our formalization.

### Equivalence of Automata

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

## 4.2 Equivalence of DFA and NFA

Deterministic and non-deterministic finite automata are equally expressive. One direction is trivial since every DFA can be seen as an NFA.

**Theorem 4.2.1.** *Let  $A = (Q, s, F, \delta)$  be a DFA. Then  $A$  is also an NFA.*

*Proof.*  $(Q, s, F, \delta)$  fulfills the definition of an NFA. Thus,  $A$  is an NFA.  $\square$

Formally, we have to construct an equivalent NFA, since DFAs and NFAs are different types. The construction is straight-forward.

**Definition** `dfa_to_nfa` : `nfa` :=  
`{ | nfa_s := dfa_s A;`  
`nfa_fin := dfa_fin A;`  
`nfa_step := fun x a y => y == dfa_step A x a | }.`

**Lemma** `dfa_to_nfa_correct` : `dfa_lang A =i nfa_lang dfa_to_nfa`.

We prove the other direction using the powerset construction.

**Definition 4.2.2.** *Given NFA  $A$ , we construct an equivalent DFA  $A_{det}$  in the following way:*

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

## 4 Finite Automata

The formalization of  $A_{det}$  is straight-forward. The set of states is an implicit argument of the DFA constructor and thus not shown.

**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 4.2.3.** *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 prove 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  $w' \in \mathcal{L}_Y(A_{det})$  by inductive hypothesis. With  $(X, a, Y) \in \delta_{det}$  we get  $aw' \in \mathcal{L}_X(A_{det})$ .

□

**Lemma 4.2.4.** *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.* By induction on  $w$ .

- 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 inductive 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 4.2.5.** *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 4.2.3 with  $x := s$  and  $X := s_{det}$ .

“ $\supseteq$ ” From Lemma 4.2.4 with  $X = s_{det}$  we get  $\mathcal{L}_{s_{det}}(A_{det}) \subseteq \mathcal{L}_s(A)$ , which proves the claim. □

## 4 Finite Automata

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

**Lemma** `nfa_to_dfa_aux2` ( $x$ :  $A$ )  $w$  ( $X$ : `nfa_to_dfa`):  
 $x \setminus \text{in } X \rightarrow \text{nfa\_accept } A \times w \rightarrow \text{dfa\_accept nfa\_to\_dfa } X \ w.$

**Lemma** `nfa_to_dfa_aux1` ( $X$ : `nfa_to_dfa`)  $w$ :  
 $\text{dfa\_accept nfa\_to\_dfa } X \ w \rightarrow [\text{exists } x, (x \setminus \text{in } X) \ \&\& \ \text{nfa\_accept } A \times w].$

**Lemma** `nfa_to_dfa_correct` : `nfa_lang A =i dfa_lang nfa_to_dfa` .

### 4.3 Connected Components

Finite automata can have isolated subsets of states that are not reachable from the starting state. Removing these states does not change the language. 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 4.3.1.** *Let  $A = (Q, s, F, \delta)$  be a DFA. We define `reachable1` such that for all  $x$  and  $y$ ,  $(x, y) \in \text{reachable1} \iff \exists a, (x, a, y) \in \delta$ . 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  $A_c$ :*

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

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 which ensures that 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`  $x \ a$  :  $x \setminus \text{in } \text{reachable} \rightarrow \text{dfa\_step } A1 \ x \ a \setminus \text{in } \text{reachable}$  .

**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 |}` .

**Lemma 4.3.2.** *For every state  $x \in \text{reachable}$  we have that*

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

*Proof.* “ $\subseteq$ ” Trivial. “ $\supseteq$ ” By induction on  $w$ .

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

□

**Theorem 4.3.3.** *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  $s \in \text{reachable}$ . We use Lemma 4.3.2 with  $x := s$  to prove the claim. □

The formalization of Lemma 4.3.2 and Theorem 4.3.3 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 prove a characteristic property of  $A_c$ . We will need this property of  $A_c$  in Chapter 5.

**Definition 4.3.4.** *A **representative** of a state  $x$  is a word  $w$  such that the unique run of  $w$  on  $A_c$  ends in  $x$ .*

**Lemma 4.3.5.** *There is 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 in  $x$ . We build the representative by extracting the edges of the path and building a word from those. □

**Lemma** `dfa_connected_repr x :`  
`exists w, last (dfa_s dfa_connected) (dfa_run dfa_connected w) = x.`

## 4.4 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.

## 4 Finite Automata

**Theorem 4.4.1.** *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 4.3.3 we have  $\mathcal{L}(A) = \mathcal{L}(A_c)$ . Therefore, it suffices to show

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

“ $\Rightarrow$ ” 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 4.3.5 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  $x \notin F_c$ .

“ $\Leftarrow$ ” 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,  $w \notin \mathcal{L}(A_c)$ .  $\square$

Thus, emptiness is decidable.

**Definition** `dfa_lang_empty := #|dfa_fin dfa_connected| == 0.`

**Lemma** `dfa_lang_empty_correct:`  
`reflect (dfa_lang A1 == pred0)`  
`dfa_lang_empty.`

## 4.5 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 4.5.1.** *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.

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

$\square$

Thus, equivalence is decidable.

**Definition** `dfa_sym_diff A1 A2 :=`  
`dfa_disj (dfa_conj A1 (dfa_compl A2)) (dfa_conj A2 (dfa_compl A1)).`

**Definition** `dfa_equiv A1 A2 := dfa_lang_empty (dfa_sym_diff A1 A2).`

**Lemma** `dfa_equiv_correct A1 A2:`  
`dfa_equiv A1 A2 <=> dfa_lang A1 == dfa_lang A2.`

## 4.6 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.

**Theorem 4.6.1.** *Let  $r$  be an extended regular expression on  $\Sigma$ . Then we can give DFA  $A$  such that*

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

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

We show our implementation for Void, Not, and Conc. We also give a short overview of the automaton corresponding to Star.

### 4.6.1 Void

**Definition 4.6.2.** *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 4.6.3.** *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`  $x \ w: \sim \sim \text{dfa\_accept } \text{dfa\_void } x \ w.$

### 4.6.2 Not

**Definition 4.6.4.** *Given DFA  $A = (Q, s, F, \delta)$ , the complement automaton  $A_\neg$  is constructed by swapping accepting and non-accepting states, i.e.*

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

**Lemma 4.6.5.** *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.  $\mathcal{L}_x(A_\neg) = \Sigma^* \setminus \mathcal{L}_x(A)$*

*Proof.* By induction on  $w$ . For  $w = \varepsilon$  we have  $\varepsilon \in \mathcal{L}_x(A_\neg) \iff \varepsilon \in \mathcal{L}_x(A)$  from  $x \in F \iff x \notin Q \setminus F$ . For  $w = aw'$  we get  $(y, a, x) \in \delta$ . By inductive 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 4.6.6.**  *$A_\neg$  accepts the complement language of  $A$ , i.e.  $\mathcal{L}(A_\neg) = \Sigma^* \setminus \mathcal{L}(A)$ .*

*Proof.* This follows directly from Lemma 4.6.5 with  $x := s$ . □



### 4.6.3 Conc

The most common approach to build the concatenation automaton is to connect the final states of the first automaton to the starting state of the second automaton by an  $\varepsilon$ -transition. We do not allow  $\varepsilon$ -transitions in our automata. The reason for this is that we do not want to lose the direct correspondence of the length of the word to the length of its run on an automaton. Thus, in order to build the concatenation automaton, we duplicate all edges from the starting state of the second automaton and add them to all final states of the first automaton. Since the final states may already have edges with the same labels, we chose to implement this operation on NFAs.

**Definition 4.6.7.** *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:*

$$\begin{aligned} Q_{Conc} &:= Q_1 \cup Q_2 \\ s_{Conc} &:= s_1 \\ F_{Conc} &:= \begin{cases} F_2 & \text{if } s_2 \notin F_2 \\ F_2 \cup F_1 & \text{if } s_2 \in F_2 \end{cases} \\ \delta_{Conc} &:= \delta_1 \cup \delta_2 \cup \{(x, a, y) \mid x \in Q_1, y \in Q_2, (s_2, a, y) \in \delta_2\} \\ A_{Conc} &:= (Q_{Conc}, s_{Conc}, F_{Conc}, \delta_{Conc}). \end{aligned}$$

**Definition** `nfa_conc : nfa :=`  
`{ | nfa_s := inl _ (nfa_s A1);`  
`nfa_fin := [fun x =>`  
`match x with`  
`| inl x => nfa_fin A1 x && nfa_fin A2 (nfa_s A2)`  
`| inr x => nfa_fin A2 x`  
`end];`  
`nfa_step := fun x a y =>`  
`match x,y with`  
`| inl x, inl y => nfa_step A1 x a y`  
`| inl x, inr y => nfa_fin A1 x && nfa_step A2 (nfa_s A2) a y`  
`| inr x, inr y => nfa_step A2 x a y`  
`| inr x, inl y => false`  
`end |}.`

Before we prove the correctness of  $A_{Conc}$ , we need a number of auxiliary lemmas.

**Lemma 4.6.8.** *Every run of  $A_2$  can be mapped to a run in  $A_{Conc}$ .*

*Proof.* Let  $\sigma$  be a run starting in  $x$  for  $w \in \Sigma^*$  on  $A_2$ . By induction on  $\sigma$ .

1. For  $\sigma = x$  we have  $w = \varepsilon$ . Therefore, we have that  $\sigma$  is also a run starting in  $x$  for  $\varepsilon$  on  $A_{Conc}$ .
2. For  $\sigma = xy\sigma'$  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 inductive hypothesis, we have that  $y\sigma'$  is a run for  $w'$  starting in  $y$  on  $A_{Conc}$ . Thus,  $xy\sigma'$  is a run for  $aw'$  starting in  $x$  on  $A_{Conc}$ .

□

**Lemma** `nfa_conc_cont x xs w:`

`nfa_run A2 x xs w`  
 $\rightarrow$  `nfa_run nfa_conc (inl _ x) (map (@inr A1 A2) xs) w.`

The next lemma shows that, in  $A_{Conc}$ , the final states of  $A_1$  have all transitions that the starting state of  $A_2$  also has. Consequently, they accept the same words.

**Lemma 4.6.9.** *Let  $x \in F_1$ . Let  $w \in \mathcal{L}(A_2)$ . Then  $w \in \mathcal{L}_x(A_{Conc})$ .*

*Proof.* By induction on  $w$ .

1. For  $w = \varepsilon$  we have get  $s \in F_2$  by  $\varepsilon \in \mathcal{L}(A_2)$  and, thus,  $x \in F_{Conc}$  by definition.
2. For  $w = aw'$  we have  $y \in Q_2$  such that  $(s, a, y) \in \delta_2$  and thus  $(x, a, y) \in \delta_{Conc}$ . We also have  $w' \in \mathcal{L}_y(A_2)$  and thus, by Lemma 4.6.8,  $w' \in \mathcal{L}_y(A_{Conc})$ . Thus,  $aw' \in \mathcal{L}_x(A_{Conc})$ .

□

**Lemma** `nfa_conc_fin1 x1 w:`

`nfa_fin A1 x1  $\rightarrow$`   
`nfa_lang A2 w  $\rightarrow$`   
`nfa_accept nfa_conc (inl _ x1) w.`

The following lemma is one direction of the proof of correctness of  $A_{Conc}$ .

**Lemma 4.6.10.** *Let  $x \in Q_1$ ,  $w_1 \in \mathcal{L}_x(A_1)$ , and  $w_2 \in \mathcal{L}(A_2)$ . Then  $w_1w_2 \in \mathcal{L}_x(A_{Conc})$ .*

*Proof.* By induction on  $w_1$ .

1. For  $w_1 = \varepsilon$  we get  $x \in F_1$  and thus, by Lemma 4.6.9, the claim follows.
2. For  $w_1 = aw'_1$  we get  $(x, a, y) \in \delta_1$  and thus  $(x, a, y) \in \delta_{Conc}$ . By inductive hypothesis, the claim follows.

□

**Lemma** `nfa_conc_aux2 x w1 w2:`

`nfa_accept A1 x w1  $\rightarrow$`   
`nfa_lang A2 w2  $\rightarrow$`   
`nfa_accept nfa_conc (inl _ x) (w1 ++ w2).`

The next lemma constitutes the other direction. Its statement is very general, even though we will only need one of the two cases for the proof of correctness of  $A_{Conc}$ . However, with the second case, there is no straight-forward inductive proof.

**Lemma 4.6.11.** *Let  $x \in Q_{Conc}$ . Let  $w \in \mathcal{L}_x(A_{Conc})$ . Then, either*

$$x \in Q_1 \wedge \exists w_1. \exists w_2. w = w_1w_2 \wedge w_1 \in \mathcal{L}_x(A_1) \wedge w_2 \in \mathcal{L}(A_2), \quad (*)$$

or

$$x \in Q_2 \wedge w \in \mathcal{L}_x(A_2). \quad (**)$$

## 4 Finite Automata

*Proof.* By induction on  $w$ .

1. For  $w = \varepsilon$  we get either  $x \in F_1$  and  $s_2 \in F_2$  or  $x \in F_2$ . In the first case, we need to prove (\*), which we do by choosing  $w_1 := \varepsilon$  and  $w_2 := \varepsilon$ . In the second case, we need to prove  $\varepsilon \in \mathcal{L}_x(A_2)$  and thus  $x \in F_2$  which we have by assumption.
2. For  $w = aw'$  we get  $y$  such that  $(x, a, y) \in \delta_{Conc}$  and  $w' \in \mathcal{L}_y(A_{Conc})$ . We are left with four cases, depending on the origin of  $x$  and  $y$ .
  - a) For  $x, y \in Q_2$  we have  $(x, a, y) \in Q_2$  and claim (\*\*) follows.
  - b) For  $x, y \in Q_1$  we prove (\*). By inductive hypothesis we get  $w_1$  and  $w_2$  such that (\*) holds for  $y$ .
  - c) For  $x \in Q_1$  and  $y \in Q_2$  the claim follows with  $w_1 := \varepsilon$  and  $w_2 := aw'$  by inductive hypothesis.
  - d) For  $x \in Q_2$  and  $y \in Q_1$  we have  $(x, a, y) \in \delta_{Conc}$ , which is a contradiction.

□

**Lemma** `nfa_conc_aux1` X w :  
`nfa_accept nfa_conc X w ->`  
**match** X with  
| inl x => **exists** w1, **exists** w2, (w == w1 ++ w2) && (nfa\_accept A1 x w1) && nfa\_lang A2 w2  
| inr x => nfa\_accept A2 x w  
**end.**

**Corollary 4.6.12.** *The language of  $A_{Conc}$  is the concatenation of the languages of  $A_1$  and  $A_2$ , i.e.  $\mathcal{L}(A_{Conc}) = \mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$ .*

*Proof.* Follows directly from Lemma 4.6.10 and Lemma 4.6.11. □

**Lemma** `nfa_conc_correct`: `nfa_lang nfa_conc =i conc (nfa_lang A1) (nfa_lang A2)`.

### 4.6.4 Star

The most common construction for the star automaton works by adding the starting state to the set of final states and connecting all final states to the starting state by  $\varepsilon$ -transitions.

Again, our construction differs from this. First, we construct an automaton that accepts the Kleene closure of the language of the given automaton, excluding the empty word, which we call `nfa_repeat`. The reason for this is that we can easily construct this automaton much in the same way we constructed the concatenation automaton.

We duplicate all edges from the starting state and add them to the final states. The resulting automaton accepts the Kleene closure of the language of the given automaton, but not the empty word. Since we have already constructed an automaton that accepts the empty word, and a disjunction operation on automata, we simply combine those with our newly constructed automaton to form the star automaton.

**Definition**  $\text{nfa\_star} := (\text{dfa\_disj } \text{dfa\_eps } (\text{nfa\_to\_dfa } \text{nfa\_repeat })).$

**Lemma**  $\text{nfa\_star\_correct}: \text{dfa\_lang } \text{nfa\_star} = \text{i star } (\text{nfa\_lang } A1).$

We give a procedure to build an equivalent DFA for every extended regular expression and prove it correct. Note that the operations are named after the type of arguments they take, i.e.  $\text{nfa\_star}$  takes an NFA but returns a DFA, whereas  $\text{nfa\_conc}$  expects *and* returns NFAs.

**Fixpoint**  $\text{re\_to\_dfa } (r: \text{regular\_expression } \text{char}): \text{dfa char} :=$   
 $\text{match } r \text{ with}$   
 $\quad | \text{Void} \Rightarrow \text{dfa\_void char}$   
 $\quad | \text{Eps} \Rightarrow \text{dfa\_eps char}$   
 $\quad | \text{Dot} \Rightarrow \text{dfa\_dot char}$   
 $\quad | \text{Atom } a \Rightarrow \text{dfa\_char char } a$   
 $\quad | \text{Star } s \Rightarrow \text{nfa\_star } (\text{dfa\_to\_nfa } (\text{re\_to\_dfa } s))$   
 $\quad | \text{Plus } s \ t \Rightarrow \text{dfa\_disj } (\text{re\_to\_dfa } s) (\text{re\_to\_dfa } t)$   
 $\quad | \text{And } s \ t \Rightarrow \text{dfa\_conj } (\text{re\_to\_dfa } s) (\text{re\_to\_dfa } t)$   
 $\quad | \text{Conc } s \ t \Rightarrow \text{nfa\_to\_dfa } (\text{nfa\_conc } (\text{dfa\_to\_nfa } (\text{re\_to\_dfa } s)) (\text{dfa\_to\_nfa } (\text{re\_to\_dfa } t))))$   
 $\quad | \text{Not } s \Rightarrow \text{dfa\_compl } (\text{re\_to\_dfa } s)$   
 $\text{end.}$

**Lemma**  $\text{re\_to\_dfa\_correct } r: \text{dfa\_lang } (\text{re\_to\_dfa } r) = \text{i r}.$

## 4.7 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 DFAs  $A_1$  and  $A_2$  as above. Based on our decision procedure for the equivalence of DFAs, we only need check if  $A_1$  and  $A_2$  are equivalent.

**Theorem 4.7.1.** *Let  $r, s$  be regular expressions on  $\Sigma$  and  $A_1, A_2$  their corresponding, equivalent automata. We then have that*

$$\mathcal{L}(r) = \mathcal{L}(s) \iff \mathcal{L}(A_1) = \mathcal{L}(A_2).$$

*Proof.* Follows directly from 4.6.1 and 4.5.1. □

Thus, equivalence is decidable.

**Definition**  $\text{re\_equiv } r \ s := \text{dfa\_equiv } (\text{re\_to\_dfa } r) (\text{re\_to\_dfa } s).$

**Lemma**  $\text{re\_equiv\_correct } r \ s: \text{re\_equiv } r \ s \iff r = \text{i s}.$

## 4.8 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**” [12] (reformulated in [17]), which works by sequentially building up regular expressions on the edges between states. In

## 4 Finite Automata

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 formalization of this method is rather complicated, due to the fact that we need generalized finite automata, i.e., finite automata whose edges are labeled by regular expressions. Thus, we decided against this method.

The second method is known as “**Brzowski's method**” [13] and builds upon Brzowski derivatives of regular expressions. This method is algebraic in nature and arrives at a regular expression by solving a system of linear equations on regular expressions. Every state is assigned an unknown regular expression. The intuition of these unknown regular expressions is that they recognize the words accepted in their associated state. The system is solved by substitution and Arden's lemma [3]. The regular expressions associated with the starting state recognizes the language of the automaton.

Brzowski's method requires infrastructure for algebraic equations and term rewriting, which we estimated to be a considerable overhead. Thus, we decided against this method.

The third method, which we choose for our development, is due to Kleene [21]. 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 this chapter, we assume that we are given a DFA  $(Q, s, F, \delta)$ .

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 every word 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  $X$ . Here, passing through 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 (singleton) 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 constitute all possible words that lead from  $x$  to  $y$  without passing through any state.

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.

## 4 Finite Automata

- (ii) The second part contains that part of the remainder of  $\sigma$  which contains all further occurrences of  $z$  with the exception of 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  $X \setminus \{z\}$ . 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  $X \setminus \{z\}$  with the help of the  $*$  operator.

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

In order to define  $R$  recursively, we need to pick an element  $z \in X$  if  $X \neq \emptyset$ . For this purpose, we assume an ordering on  $Q$ . We will then pick  $z \in X$  such that  $z$  is the smallest element in  $X$  with respect to this ordering.

**Definition 4.8.1.** Let  $X \subseteq Q$ . Let  $x, y \in Q$ . We define  $R$  recursively on  $|X|$ :

$$R_{x,y}^X := \begin{cases} \sum_{\substack{a \in \Sigma \\ (x,a,y) \in \delta}} a & \text{if } X = \emptyset \wedge x \neq y; \\ \sum_{\substack{a \in \Sigma \\ (x,a,y) \in \delta}} a + \varepsilon & \text{if } X = \emptyset \wedge x = y; \\ R_{x,z}^{X \setminus \{z\}} (R_{z,z}^{X \setminus \{z\}})^* R_{z,y}^{X \setminus \{z\}} + R_{x,y}^{X \setminus \{z\}} & \text{if } X \neq \emptyset \wedge z \text{ minimal in } X. \end{cases}$$

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.  $\text{nPlus}$  is  $\sum$  on regular expressions.  $\text{dfa\_step\_any}$  is the list chars of that contains all transitions from  $x$  to  $y$ .  $R0$  covers the case of  $X = \emptyset$ .

**Definition**  $\text{nPlus rs} := \text{foldr } (@\text{Plus char}) (\text{Void } \_) \text{ rs}$ .

**Definition**  $\text{dfa\_step\_any } x \ y := \text{enum } ([\text{pred } a \mid \text{dfa\_step } A \ x \ a == y])$ .

**Definition**  $R0 \ x \ y := \text{let } r := \text{nPlus } (\text{map } (@\text{Atom } \_) (\text{dfa\_step\_any } x \ y)) \text{ in}$   
 $\text{if } x == y \text{ then Plus } r \ (\text{Eps } \_) \text{ else } r$ .

**Function**  $R \ (X: \{\text{set } A\}) \ (x \ y: A) \ \{\text{measure } [\text{fun } X \Rightarrow \#|X|] \ X\} :=$   
 $\text{match } [\text{pick } z \text{ in } X] \text{ with}$   
 $\mid \text{None} \Rightarrow R0 \ x \ y$   
 $\mid \text{Some } z \Rightarrow \text{let } X' := X \setminus z \text{ in}$   
 $\text{Plus } (\text{Conc } (R \ X' \ x \ z) \ (\text{Conc } (\text{Star } (R \ X' \ z \ z)) \ (R \ X' \ z \ y))) \ (R \ X' \ x \ y)$   
 $\text{end}$ .

We now express  $\mathcal{L}(A)$  using  $R$ . 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  $R_{s,f}^Q$  to form a regular expression for  $\mathcal{L}(A)$ . The goal of this chapter is to prove the following theorem.

**Theorem 4.8.2.**  $\mathcal{L}(A)$  is recognizable by a regular expression, i.e.

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

## 4 Finite Automata

In order to prove this theorem, we will first define a predicate on words that corresponds to  $\mathcal{L}(R_{x,y}^X)$ . We call this predicate  $L_{x,y}^X$  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  $y$ .

**Definition 4.8.3.** 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  $L_{x,y}^X$  such that

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

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 `belast` to remove the last element from a sequence. Note that, mathematically, runs include the starting state. In our formalization, this is not the case. Thus, we do not need to remove the first state from a run to retrieve all states the run passes through.

**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) ]`.

We now prove properties of  $L$  that we will need for our proof of Theorem 4.8.2.

**Lemma 4.8.4.**  $L$  is monotone in  $X$ , i.e.

$$\forall X \subseteq Q, z \in X, x, y \in Q. L_{x,y}^X \subseteq L_{x,y}^{X \cup \{z\}}.$$

*Proof.* This follows directly from  $X \subset X \cup \{z\}$ . □

**Lemma** `L_monotone (X: {set A}) (x y z: A): {subset L^X x y <= L^{(z | X)} x y}`.

**Lemma 4.8.5.** The empty word is contained in  $L_{x,y}^X$  if and only if  $x = y$ .

*Proof.* This follows immediately from the definition of  $L$ . □

**Lemma** `L_nil X x y: reflect (x = y) ([::] \in L^X x y)`.

Next, we will prove that words whose run passes through a state  $z$  can be split into two words. The run of the first word will end in  $z$ , i.e. not pass through  $z$ .

**Lemma 4.8.6.** Let  $w \in \Sigma^*$ . Let  $x, z \in Q$ . Let  $\sigma$  be the run of  $w$  on  $A$  starting in  $x$ . Let  $z \in \sigma_1 \dots \sigma_{|\sigma|-1}$ . Then there exist  $w_1, w_2 \in \Sigma^*$  such that

$$w = w_1 w_2 \wedge |w_2| < |w| \wedge z \notin \sigma_1 \dots \sigma_{|w_1|-1} \wedge \sigma_{|w_1|} = z.$$

*Proof.* Let  $i$  be the first occurrence of  $z$  in  $\sigma_1 \dots \sigma_{|\sigma|-1}$  such that  $\sigma_i = z$  and  $i > 0$ . Let  $w_1 := w_0 \dots w_{i-1}$  and  $w_2 := w_i \dots w_{|w|-1}$ . The claim follows. □

## 4 Finite Automata

**Lemma**  $\text{run\_split } x \ z \ w: z \ \backslash \text{in } \text{dfa\_run}' \ A \times w \rightarrow$   
 $\text{exists } w_1, \text{ exists } w_2,$   
 $w = w_1 ++ w_2 \ \wedge$   
 $\text{size } w_2 < \text{size } w \ \wedge$   
 $z \ \backslash \text{notin } \text{belast } (\text{dfa\_run}' \ A \times w_1) \ \wedge$   
 $\text{last } x \ (\text{dfa\_run}' \ A \times w_1) = z.$

We will make use of this fact in the next lemma, which splits words in  $L^X$  into two parts, the first of which is again in  $L^X$ . This will be quintessential later, when we split words in  $L^X$  into three parts that correspond to the recursive definition of  $R^X$ .

**Lemma 4.8.7.** *Let  $X \subseteq Q$  and  $x, y, z \in Q$ . Let  $w \in L_{x,y}^{X \cup \{z\}}$ . We have that either*

$$w \in L_{x,y}^X$$

*or there exist  $w_1$  and  $w_2$  such that*

$$w = w_1 w_2 \ \wedge \ |w_2| < |w| \ \wedge \ w_1 \in L_{x,z}^X \ \wedge \ w_2 \in L_{z,y}^{X \cup \{z\}}.$$

*Proof.* We first eliminate the case of  $z \in X$ , which is solved trivially. Let  $\sigma$  be the run of  $w$  on  $A$  starting in  $x$ . We do a case distinction on  $z \in \sigma_1 \dots \sigma_{|\sigma|-1}$ .

1. For  $z \notin \sigma_1 \dots \sigma_{|\sigma|-1}$  we can easily show  $w \in L_{x,y}^X$ .
2. For  $z \in \sigma_1 \dots \sigma_{|\sigma|-1}$  we use Lemma 4.8.6 to split  $w$  in  $w_1$  and  $w_2$ . From  $w \in L_{x,y}^{X \cup \{z\}}$  and  $\sigma_{|w_1|} = z$  we immediately get  $w_2 \in L_{z,y}^{X \cup \{z\}}$ . We have  $z \notin \sigma_1 \dots \sigma_{|\sigma|-1}$ . We also have that  $X = (X \cup \{z\}) \setminus \{z\}$  from  $z \notin X$ . Thus, we get  $w_1 \in L_{x,z}^X$ . The remainder of the claim follows directly from Lemma 4.8.6.

□

Before we show that  $L^X$  respects the defining equation of  $R^X$ , we have to show that we can combine words from  $L_{x,z}^X$ ,  $(L_{z,z}^X)^*$ , and  $L_{z,y}^X$  to form a word in  $L_{x,y}^{X \cup \{z\}}$ . We prove a general concatenation lemma for  $L^X$ .

**Lemma 4.8.8.** *Let  $X \subseteq Q$ ,  $x, y \in Q$ , and  $z \in X$ . Let  $w_1 \in L_{x,z}^X$  and  $w_2 \in L_{z,x}^X$ . Then we have*

$$w_1 w_2 \in L_{x,y}^X.$$

*Proof.* By  $z \in X$ ,  $w_1 \in L_{x,y}^X$ , and  $\sigma_{|w_1|} = z$  we get  $\sigma_1, \dots, \sigma_{|w_1|} \in X$ . We also have  $\sigma_{|w_1|+1}, \dots, \sigma_{|\sigma|-2} \in X$  and  $\sigma_{|\sigma|-1} = y$ . Thus,  $w_1 w_2 \in L_{x,y}^X$ . □

**Lemma**  $\text{L\_cat } (X: \{\text{set } A\}) \times y \ z \ w_1 \ w_2:$   
 $z \ \backslash \text{in } X \rightarrow$   
 $w_1 \ \backslash \text{in } L^X \times z \rightarrow$   
 $w_2 \ \backslash \text{in } L^X \times y \rightarrow$   
 $w_1 ++ w_2 \ \backslash \text{in } L^X \times y.$



## 4 Finite Automata

**Lemma 4.8.9.** *Let  $n \in \mathbb{N}$ . Let  $w_0, \dots, w_{n-1} \in L_{z,z}^X$ . We have that*

$$w_0 \dots w_{n-1} \in L_{z,z}^{X \cup \{z\}}.$$

*Proof.* By induction on  $n$ .

1. For  $n = 0$  we have to prove  $\varepsilon \in L_{z,z}^{X \cup \{z\}}$  which holds by Lemma 4.8.5.
2. For  $n = n' + 1$  we have  $w_0 \dots w_{n-2} \in L_{z,z}^{X \cup \{z\}}$  by inductive hypothesis. We also have  $w_0, \dots, w_{n-1} \in L_{z,z}^X$  by assumption, and, thus,  $w_{n-1} \in L_{z,z}^X$ . By Lemma 4.8.8 we get  $w_0 \dots w_{n-1} \in L_{z,z}^{X \cup \{z\}}$ .

□

**Lemma**  $\text{L\_flatten } (X: \{\text{set } A\}) \text{ } z \text{ } vv: \text{ all } (L^X z \text{ } z) \text{ } vv \text{ } \rightarrow$   
 $\text{flatten } vv \text{ } \backslash \text{in } L^X(z \text{ } |: X) \text{ } z \text{ } z.$

Finally, we can show that  $L^X$  respects the defining equation of  $R^X$ . With all the lemmas we have in place now, this can now be shown with relative ease.

**Lemma 4.8.10.** *Let  $X \subseteq Q$ ,  $x, y \in Q$ , and  $z \in X$ . We have that*

$$L_{x,y}^{X \cup \{z\}} = L_{x,z}^X (L_{z,z}^X)^* L_{z,y}^X + L_{x,y}^X.$$

*Proof.* “ $\Rightarrow$ ” By induction on  $|w|$ .

1. For  $|w| = 0$  we get  $w \in L_{x,y}^X$  by Lemma 4.8.5.
2. For  $|w| > 0$  we get  $w_1$  and  $w_2$  such that  $w = w_1 w_2$ ,  $w_1 \in L_{x,z}^X$  and  $w_2 \in L_{z,y}^{X \cup \{z\}}$ . By inductive hypothesis we get

$$w_2 \in L_{z,z}^X (L_{z,z}^X)^* L_{z,y}^X \vee w_2 \in L_{z,y}^X$$

The latter gives us  $w \in L_{x,y}^{X \cup \{z\}}$  by Lemma 4.8.8. With the former, we have  $w_3$ ,  $w_4$ , and  $w_5$  such that  $w_2 = w_3 w_4 w_5$ ,  $w_3 \in L_{z,z}^X$ ,  $w_4 \in (L_{z,z}^X)^*$  and  $w_5 \in L_{z,y}^X$ . We merge  $w_3$  and  $w_4$  such that  $w_3 w_4 \in (L_{z,z}^X)^*$ . This gives us  $w_2 \in (L_{z,z}^X)^* L_{z,y}^X$ . Thus,  $w_1 w_2 \in L_{x,z}^X (L_{z,z}^X)^* L_{z,y}^X$ .

“ $\Leftarrow$ ” We have  $w_1 \in L_{x,z}^X$ ,  $w_2 \in (L_{z,z}^X)^*$ , and  $w_3 \in L_{z,y}^X$ . By Lemma 4.8.9 we get  $w_2 \in L_{z,z}^{X \cup \{z\}}$ . Thus,  $w_1 w_2 w_3 \in L_{x,y}^{X \cup \{z\}}$  by Lemma 4.8.8. □

**Lemma**  $\text{L\_rec } (X: \{\text{set } A\}) \text{ } x \text{ } y \text{ } z:$

$$L^X(z \text{ } |: X) \text{ } x \text{ } y = \text{plus } (\text{conc } (L^X x \text{ } z) \text{ } (\text{conc } (\text{star } (L^X z \text{ } z)) \text{ } (L^X z \text{ } y))) \\ (L^X x \text{ } y).$$

All that remains to complete the proof of Lemma 4.8.2 is a proof of  $L_{x,y}^X = \mathcal{L}(R_{x,y}^X)$ .

**Lemma 4.8.11.** *Let  $X \subseteq Q$ . Let  $x, y \in Q$ . We have that*

$$L_{x,y}^X = R_{x,y}^X.$$

*Proof.* By induction on  $|X|$ .

1. For  $|X| = 0$ , the claim follows immediately from the definitions of  $L$  and  $R$ .
2. For  $|X| = n + 1$  for some  $n \in \mathbb{N}$  we get  $\exists z \in X$  and thus

$$R_{x,y}^X = R_{x,z}^{X \setminus \{z\}} (R_{z,z}^{X \setminus \{z\}})^* R_{z,y}^{X \setminus \{z\}} + R_{x,y}^{X \setminus \{z\}}.$$

By Lemma 4.8.10 we also know that

$$L_{x,y}^X = L_{x,z}^{X \setminus \{z\}} (L_{z,z}^{X \setminus \{z\}})^* L_{z,y}^{X \setminus \{z\}} + L_{x,y}^{X \setminus \{z\}}.$$

The claim follows by inductive hypothesis. □

**Lemma** `L_R n (X: {set A}) × y: #|X| = n → L^X x y =i R^X x y.`

This concludes the proof of Theorem 4.8.2.

**Definition** `dfa_to_re : regular_expression char :=  
nPlus (map (R^setT (dfa_s A)) (enum (dfa_fin A))).`

**Lemma** `dfa_to_re_correct: dfa_lang A =i dfa_to_re .`

In preparation of the final result of this chapter, we prove that  $R$  is a standard regular expression.

**Lemma 4.8.12.** *Let  $X \subseteq Q$  and  $x, y \in Q$ .  $R_{x,y}^X$  is a standard regular expression.*

*Proof.* All regular expressions used in  $R$  are standard. □

**Lemma** `R_standard (X: {set A}) × y: standard char (R^X x y).`

The proof of Theorem 4.8.2 was by far the most technically involved proof presented in this thesis. The mathematical content is rather straight-forward and intuitive. However, especially due to the need for the `allbutlast` predicate, the implementation contains a lot of quite general infrastructure. In fact, a little more than one third of the implementation (200 out of 600 lines) is taken up by `allbutlast`.

### 4.8.1 Extended Regular Expressions to Regular Languages

We can now prove that extended regular expressions are equivalent to regular languages. We show that  $R$  is a standard regular expression, which are a subset of extended regular expression which corresponds to the definition of regular languages.

**Theorem 4.8.13.** *Extended regular expressions and regular languages are equally expressive, i.e. for all  $L$ , we have  $L$  is a regular language if and only if there exists an extended regular expression that recognizes  $L$ .*

*Proof.* “ $\Rightarrow$ ” Let  $L$  be regular. Then, by definition of regular languages, there is an equivalent standard regular expression.

“ $\Leftarrow$ ” Let  $r$  be an extended regular expression. By Theorem 4.6.1 we have DFA  $A$  such that  $\mathcal{L}(A) = \mathcal{L}(r)$ . By Theorem 4.8.2 we have regular expression  $s$  such that  $\mathcal{L}(s) = \mathcal{L}(A)$ . By Lemma 4.8.12,  $s$  is a standard regular expression. Thus, by definition of standard regular expressions,  $\mathcal{L}(r)$  is a regular language.  $\square$

**Definition** `ext_re_to_std_re (r: regular_expression char) := dfa_to_re (re_to_dfa r).`

**Lemma** `ext_re_to_std_re_standard r: standard char ( ext_re_to_std_re r).`

**Lemma** `ext_re_to_std_re_correct r: ( ext_re_to_std_re r) =i r.`

## Remarks

All in all, our formalization of the results presented in this chapter add up to 1,700 lines of code. About one half of that is due to the construction of finite automata. Although most of these constructions are straight-forward, proving them correct requires a lot of attention to detail. Additionally, we opted to give a corresponding automaton for every *extended* regular expression, which, of course, adds to the expected size of the development.