Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science

Bachelor's Thesis

Constructive Formalization of Regular Languages

submitted by
Jan-Oliver Kaiser

submitted on
October 11th, 2012

Supervisor
Prof. Dr. Gert Smolka

Advisors
Christian Doczkal, M.Sc.
Prof. Dr. Gert Smolka

Reviewers
Prof. Dr. Gert Smolka
Prof. Bernd Finkbeiner, Ph.D.

Abstract

Our goal is to give a concise formalization of the equivalence between regular expressions, finite automata and the Myhill-Nerode characterization. We give procedures to convert between these characterizations and prove their correctness. Our development is done in the proof assistant COQ. We make use of the SSREFLECT plugin which provides support for finite types and other useful infrastructure for our purpose. The entire development consists of approximately 2700 lines of code.

Acknowledgements

I would like to thank my advisor Christian Doczkal for the countless hours spent guiding me and helping me overcome the difficulties I encountered both with the formalization and the process of writing this thesis. His helpful advice and frank criticism were invaluable to me.

I would also like to thank my supervisor Prof. Gert Smolka for introducing me to field of formal logic and providing me with such a challenging and rewarding topic for my bachelor's thesis. I am also very thankful for his continued support during the entire project.

Furthermore, I would like to thank Prof. Gert Smolka and Prof. Finkbeiner for reviewing this thesis.

I would also like to thank everyone else who has helped me with this thesis. Special thanks to Andrea Fischer and Yannick Baron, who helped me greatly.

Contents

1 Introduction

Our goal is to give a concise formalization of the theory of regular languages. We include several different characterizations of regular languages and prove them equivalent. These include regular expressions, finite automata and the characterizations usually combined in the Myhill-Nerode theorem. All our proofs are constructive and, thus, constitute procedures to convert between these characterizations. Our formalization also includes decisions procedures for the equivalence of finite automata and regular expressions.

Regular languages are a well-studied class of formal languages. In their current form, they were first studied by Kleene [21], who introduced regular expressions. The concept of deterministic finite automata was introduced before Kleene's invention of regular expressions by Huffman [19] and Moore [25]. Rabin and Scott later introduced the concept of non-deterministic finite automata [27], for which they were given the Turing award [4].

We take classical proofs from [22] and translate them to our constructive setting. We employ the proof assistant Coq [24] for our formalization. Additionally, we make use of the SSReflect plugin [16]. SSReflect provides support for finite types which we need for finite automata. It also comes with a lot of useful, general infrastructure for our purpose. For every lemma and theorem proven in this thesis, we provide corresponding statements from the Coq development. Our development does not depend on axioms.

One of the most interesting parts of the formalization was to find a suitable representation of quotient types in Coq, which has no notion of quotient types. Our approach seems to work very well for our purpose.

1.1 Related work

There have been many publications on formalizations of the theory of regular languages in recent years. Most of them investigate decidability of equivalence of regular expressions, often with a focus on automatically deciding Kleene algebras.

Coquand and Siles develop a decision procedure for equivalence of regular expressions [15] on the basis of Brzozowski derivatives [13] in Coq (using SSReflect) with the goal of providing a practically executable tactic on top of the decision procedure. Their development weighs in at 7,500 lines of code, 700 of which serve as the basis of our formalization.

Krauss and Nipkow give a decision procedure for equivalence of regular expressions in Isabelle/HOL [23]. Their development is very concise with just over 1,000 lines of code. Being interested only in a correct (and efficient) tactic for deciding equivalences, they did not prove completeness and termination.

Another decision procedure for equivalence of regular expressions is developed by Braibant and Pous [11], with the goal of deciding Kleene algebras in Coq. Their formalization is based on matrices and weighs in at 19,000 lines of code. It encompasses finite automata, regular expressions and the Myhill-Nerode theorem.

Moreira, Pereira and Sousa give a decision procedure for equivalence of regular expressions in Coq[26]. Their development is based on Antimirov's partial derivatives of regular expressions [2] and contains a refutation step to speed up inequality checking. It consists of 19,000 lines of code.

Asperti formalizes a decision procedure for equivalence of regular expressions [5] based on the notion of pointed regular expressions [6]. This development was done in the Matita proof assistant [7]. It weighs in at 3,400 lines of code.

There is also a paper by Wu, Zhang and Urban on formalizing the Myhill-Nerode theorem using only regular expressions and not, as is commonly done, finite automata [29]. The authors state that this unusual choice stems, at least partly, from the restrictions of Isabelle/HOL (and similar HOL-based theorem provers). In particular, the fact that Isabelle/HOL does not allow for quantification over types prevents straight-forward formalizations of finite automata. Their development consists of roughly 2,000 lines of code.

1.2 Contributions

Our formalization is done in constructive type theory. Thus, all our proofs are algorithms that are executable in theory. However, our focus is solely on clarity and simplicity. As a result, the algorithms and procedures given in this thesis are very close to the mathematical definitions in [22], but not executable in practice.

Our development shows that Coq (particularly with SSReflect) is well suited for this kind of formalization. Furthermore, we have also developed a new characterization derived from the Nerode relation and proven it equivalent to all other characterizations. Our development weighs in at about 3,500 lines of code.

1.3 Outline

In Chapter 2 we give a brief introduction to Coq and SSReflect and introduce concepts that are relevant to our formalization.

In Chapter 3 we give basic definitions (words, languages, etc.). We also introduce decidable languages, regular languages and regular expressions. Furthermore, we prove the decidability of regular languages.

In Chapter 4 we introduce finite automata. We prove the equivalence of deterministic and non-deterministic finite automata. We also give a procedure to remove unreachable states from deterministic finite automata. Furthermore, we prove decidability of emptiness and equivalence of finite automata. Finally, we prove that regular expressions and finite automata are equally expressive.

In Chapter 5 we introduce the Myhill-Nerode theorem. We give three different characterizations of regular languages based on the Myhill-Nerode theorem and prove them all equally expressive to finite automata.

2 Coq and SSReflect

We decided to employ the Small Scale Reflection Extension [16] (**SSReflect**[1]) for the **Coq**[2] proof assistant [24]. The most important factors in this decision were SSReflect's excellent support for finite types, list operations and graphs. SSReflect also introduces an alternative scripting language that can often be used to shorten the bookkeeping overhead of proofs considerably.

2.1 Coq

The Coq manual [24] states the following about Coq: The Coq system is designed to develop mathematical proofs, and especially to write formal specifications, programs and to verify that programs are correct with respect to their specification. It provides a specification language named Gallina. Terms of Gallina can represent programs as well as properties of these programs and proofs of these properties. Using the so-called Curry-Howard isomorphism, programs, properties and proofs are formalized in the same language, which is a λ-calculus with a rich type system called Calculus of Inductive Constructions. All logical judgments in Coq are typing judgments. The very heart of the Coq system is the type-checking algorithm that checks the correctness of proofs, i.e that a program complies to its specification. Coq also provides an interactive proof assistant to build proofs using specific programs called tactics.

2.2 SSReflect

SSReflect is a set of extensions to the proof scripting language of the Coq proof assistant. They were originally developed to support small-scale reflection. However, most of them are of quite general nature and improve the functionality of Coq in most basic areas such as script layout and structuring, proof context management and rewriting [16].

SSReflect comes with an extensive library covering many mathematical concepts. In fact, we barely scratch the surface of the library in this development. The interested reader may convince herself/himself of the sheer size of the library at `http://coqfinitgroup.gforge.inria.fr/`.

[1] `http://www.msr-inria.inria.fr/Projects/math-components`
[2] `http://coq.inria.fr/`

2.2.1 Boolean Reflection

SSReflect offers boolean reflections for decidable propositions. A term of type reflect P b is a proof of the equivalence of the boolean statement b and the proposition P. SSReflect has built-in support to change from boolean to propositional statements if they are equivalent. This allows us to always assume the most convenient perspective in our proofs.

2.2.2 Boolean Predicates

SSReflect has special type for boolean predicates, pred T := T −> bool, where T is a type. We make use of SSReflect's syntax to specify boolean predicates. This allows us to specify predicates in a way that resembles set-theoretic notation, e.g. [pred x | <boolean expression in x>]. Furthermore, we can use the functions pred1 and pred0 to specify the singleton predicate and the empty predicate, respectively. The complement of a predicate can be written as [predC p]. The syntax for combining predicates is [pred? p1 & p2], with ? being one of U (union), I (intersection) or D (difference). For predicates given in such a way, we write y \in p to express that y fulfills p. There is also syntax for the preimage of a predicate under a function which can be written as [preim f of p].

There are also applicative (functional) versions of of predC, predU, predI, predD, which are functions that take predicates as arguments and return predicates. pred1 x represents the predicate [pred y | y == x].

2.2.3 Finite Types

The most important feature of SSReflect for our purpose are finite types. Finite types are types that have a finite number of elements. The type of finite types is finType. SSReflect offers the following operations on finite types. We compute the number of elements for which a predicate p on a finite type returns true with #|p|. Additionally, we write #|F| for the number of elements in the finite type F. SSReflect also provides boolean versions of the universal and existential quantifiers on finite types, **forallb** and **existsb**. enum F gives a list of all items of the finite type F. Finite types are closed under basic operations such as option and sum.

We can also create finite types from lists. Instances of these types can be specified with the SeqSub constructor, which takes as arguments an element of the list and a proof that this element is contained in the list.

2.2.4 Finite Sets

SSReflect also supports finite sets, which are sets on finite types. Finite sets themselves are finite types, which enables us to use them in the construction of finite automata. Additionally, they provide the syntax for computing the number of elements in them. The type of a finite set is {set T}, where T is a finite type. The function set1 constructs a singleton set containing its first argument. The type of the resulting finite set can be inferred from the argument. The union over finite sets can be expressed by `\bigcup_(x | P x) F`, which corresponds to the mathematical notation $\bigcup_{x \in X, Px} F$, with X being the (finite) type of x.

2.2.5 Equality

We can use `f =1 g` to express that the functions `f` and `g` agree in all arguments. If we regard `f` and `g` as sets, we can write `f =i g`, which is defined as `forall x, x \in f = x \in g`. Coq's equality = is intensional, which means that even if we have `f =1 g`, we are not, in general, be able to prove `f = g`. Thus, we use `=1` or `=i` in Coq, when we write = mathematically. This expresses the notion of extensional equality of classical mathematics.

3 Decidable Languages

We give basic definitions for languages, operators on languages and, finally, regular languages. We provide the corresponding formalizations from our development and prove their correctness.

3.1 Definitions

We closely follow the definitions from [18]. An **alphabet** Σ is a finite set of symbols. A **word** w is a finite sequence of symbols chosen from some alphabet. We use $|w|$ to denote the **length** of a word w. ε denotes the empty word. Given two words $w_1 = a_1 \cdots a_n$ and $w_2 = b_1 \cdots b_m$, the **concatenation** of w_1 and w_2 is defined as $a_1 \cdots a_n b_1 \cdots b_m$ and denoted $w_1 \cdot w_2$ or just $w_1 w_2$. A **language** is a set of words. The **residual language** of a language L with respect to symbol a is the set of words u such that av is in L. The residual is denoted $res_a(L)$. We define Σ^k to be the **set of words of length k**. The **set of all words** over an alphabet Σ is denoted Σ^*, i.e.,$\Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k$. A language L is **decidable** if and only if there exists a boolean predicate that decides membership in L. We will only deal with decidable languages from here on. Throughout the remaining document, we will assume a fixed alphabet Σ.

We employ finite types to formalize alphabets. In most definitions, alphabets will not be made explicit. However, the same name and type will be used throughout the entire development. Words are formalized as sequences over the alphabet. Decidable languages are represented by functions from *word* to *bool*.

```
Variable char: finType.
Definition word := seq char.
Definition language := pred word.
Definition residual x L : language := [preim cons x of L].
```

3.1.1 Operations on Languages

We will later introduce a subset of the decidable language that is based on the following operations. For every operator, we will prove the decidabality of the resulting language.

The **concatenation** of two languages L_1 and L_2 is denoted $L_1 \cdot L_2$ and is defined as the set of words $w = w_1 w_2$ such that w_1 is in L_1 and w_2 is in L_2. The **Kleene closure** of a language L is denoted L^* and is defined as the set of words $w = w_1 \cdots w_k$ such that $w_1 \ldots w_k$ are in L. Note that $\varepsilon \in L^*$ $(k = 0)$. We define the **complement** of a language L as $L \backslash \Sigma^*$, which we write as $\neg L$. Furthermore, we make use of the standard set operations **union** and **intersection**.

For our Coq development, we take Coquand and Siles's [15] implementation of these operators. plus and prod refer to union and intersection, respectively. Additionally, we also introduce the singleton languages (atom), the empty language (void) and the language containing only the empty word (eps).

```
Definition conc L1 L2 : language :=
  fun v => [ exists i : 'I_(size v).+1, L1 (take i v) && L2 (drop i v) ].
Definition star L : language :=
  fix star v := if v is x :: v' then conc (residual x L) star v' else true.
Definition compl L : language := predC L.
Definition plus L1 L2 : language := [predU L1 & L2].
Definition prod L1 L2 : language := [predI L1 & L2].
Definition atom x : language := pred1 [:: x].
Definition void : language := pred0.
Definition eps : language := pred1 [::].
```

The definition of conc is based on a characteristic property of the concatenation of two languages. The following lemma proves this property.

Lemma 3.1.1. *Let $L_1, L_2, w = a_1 \cdots a_k$ be given. We have that*

$$w \in L_1 \cdot L_2 \iff \exists n \in \mathbb{N}. 0 < n \leq k \wedge a_1 \cdots a_{n-1} \in L_1 \wedge a_n \cdots a_k \in L_2.$$

Proof. "$\Rightarrow$" From $w \in L_1 \cdot L_2$ we have w_1, w_2 such that $w = w_1 w_2 \wedge w_1 \in L_1 \wedge w_2 \in L_2$. We choose $n := |w_1| + 1$. We then have that $a_1 \cdots a_{n-1} = a_1 \cdots a_{|w_1|} = w_1$ and $w_1 \in L_1$ by assumption. Similarly, $a_n \cdots a_k = a_{|w_1|+1} \cdots a_k = w_2$ and $w_2 \in L_2$ by assumption.
"$\Leftarrow$" We choose $w_1 := a_1 \cdots a_{n-1}$ and $w_2 := a_n \cdots a_k$. By assumption we have that $w = w_1 w_2$. We also have that $a_1 \cdots a_{n-1} \in L_1$ and $a_n \cdots a_k \in L_2$. It follows that $w_1 \in L_1$ and $w_2 \in L_2$. □

```
Lemma concP : forall {L1 L2 v},
  reflect (exists2 v1, v1 \in L1 & exists2 v2, v2 \in L2 & v = v1 ++ v2)
          (v \in conc L1 L2).
```

The implementation of star makes use of a property of the Kleene closure, which is that any nonempty word in L^* can be seen as the concatenation of a nonempty word in L and a (possibly empty) word in L^*. This property allows us to implement star as a structurally recursive predicate. The following lemma proves the correctness of this property.

Lemma 3.1.2. *Let $L, w = a_1 \cdots a_k$ be given. We have that*

$$w \in L^* \iff \begin{cases} a_2 \cdots a_k \in res_{a_1}(L) \cdot L^*, & \text{if } |w| > 0; \\ w = \varepsilon, & \text{otherwise.} \end{cases}$$

Proof. "$\Rightarrow$" We do a case distinction on $|w| = 0$.

1. $|w| = 0$. It follows that $w = \varepsilon$.

2. $|W| \neq 0$, i.e. $|w| > 0$. From $w \in L^*$ we have $w = w_1 \cdots w_l$ such that $w_1 \cdots w_l$ are in L. There exists a minimal n such that $|w_n| > 0$ and for all $m < n$, $|w_m| = 0$. Let $w_n = b_1 \cdots b_p$. We have that $b_2 \cdots a_p \in res_{b_1}(L)$. Furthermore, we have that $w_{n+1} \cdots w_l \in L^*$. We also have $a_1 = b_1$ and $w = a_1 \cdots a_k = w_n \cdots w_l$. Therefore, we have $a_2 \cdots a_k \in res_{a_1}(L) \cdot L^*$.

"$\Leftarrow$" We do a case distinction on the disjunction.

1. $w = \varepsilon$. Then $w \in L^*$ by definition.

2. $a_2 \cdots a_k \in res_{a_1}(L) \cdot L^*$. By Lemma 3.1.1 we have n such that $a_2 \cdots a_{n-1} \in res_{a_1}(L)$ and $a_n \cdots a_k \in L^*$. By definition of res, we have $a_1 \cdots a_{n-1} \in L$. Furthermore, we also have $a_n \cdots a_k = w_1 \cdots w_l$ such that $w_1 \dots w_l$ are in L. We choose $w_0 := a_1 \cdots a_{n-1}$. It follows that $w = w_0 w_1 \cdots w_l$ with $w_0, w_1, \cdots w_l$ in L. Therefore, $w \in L^*$.

□

The formalization of Lemma 3.1.2 connects the formalization of star to the mathematical definition. The propositional formula given here appears slightly more restrictive than our mathematical definition as it requires all words from L to be nonempty. Mathematically, however, this is no restriction.

```
Lemma starP : forall {L v},
  reflect (exists2 vv, all [predD L & eps] vv & v = flatten vv)
          (v \in star L).
```

Theorem 3.1.3. *The decidable languages are closed under concatenation, Kleene star, union, intersection and complement.*

Proof. We have already given algorithms for all operators. It remains to show that they are correct. For concatenation and the Kleene star, we have shown in Lemma 3.1.1 and Lemma 3.1.2 that the formalizations are equivalent to the mathematical definitions. The remaining operators (union, intersection, complement) can be applied directly to the result of the languages' boolean decision functions. □

3.2 Regular Languages

Definition 3.2.1. *The set of regular languages REG is defined to be exactly those languages generated by the following inductive definition:*

$$\frac{}{\emptyset \in REG} \qquad \frac{}{\{\varepsilon\} \in REG} \qquad \frac{a \in \Sigma}{\{a\} \in REG} \qquad \frac{L \in REG}{L^* \in REG}$$

$$\frac{L_1 \in REG \qquad L_2 \in REG}{L_1 \cup L_2 \in REG} \qquad \frac{L_1 \in REG \qquad L_2 \in REG}{L_1 \cdot L_2 \in REG}$$

3.2.1 Regular Expressions

Regular expressions mirror the definition of regular languages very closely.

Definition 3.2.2. *We will consider* ***extended regular expressions*** *that include negation (*Not*), intersection (*And*) and a single-symbol wildcard (*Dot*). Therefore, we have the following syntax for regular expressions:*

$$r, s := \emptyset \mid \varepsilon \mid . \mid a \mid r^* \mid r + s \mid r\&s \mid rs \mid \neg r$$

The language of an extended regular expression is defined as follows:

$$\begin{aligned} \mathcal{L}(\emptyset) &= \emptyset & \mathcal{L}(r^*) &= \mathcal{L}(r)^* \\ \mathcal{L}(\varepsilon) &= \{\varepsilon\} & \mathcal{L}(r+s) &= \mathcal{L}(r) \cup \mathcal{L}(s) \\ \mathcal{L}(.) &= \Sigma & \mathcal{L}(r\&s) &= \mathcal{L}(r) \cap \mathcal{L}(s) \\ \mathcal{L}(a) &= \{a\} & \mathcal{L}(rs) &= \mathcal{L}(r) \cdot \mathcal{L}(s) \end{aligned}$$

Definition 3.2.3. *We say that two regular expressions r and s are equivalent if and only if*

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

We will later show that equivalence of regular expressions is decidable. We take the implementation of regular expressions from Coquand and Siles's development ([15]), which is also based on SSReflect and comes with helpful infrastructure for our proofs. The semantics defined in Definition 3.2.2 can be given as a boolean function.

```
Inductive regular_expression :=
 | Void
 | Eps
 | Dot
 | Atom of symbol
 | Star of regular_expression
 | Plus of regular_expression & regular_expression
 | And of regular_expression & regular_expression
 | Conc of regular_expression & regular_expression
 | Not of regular_expression .

Fixpoint mem_reg e :=
  match e with
  | Void => void
  | Eps => eps
  | Dot => dot
  | Atom x => atom x
  | Star e1 => star (mem_reg e1)
  | Plus e1 e2 => plus (mem_reg e1) (mem_reg e2)
  | And e1 e2 => prod (mem_reg e1) (mem_reg e2)
  | Conc e1 e2 => conc (mem_reg e1) (mem_reg e2)
  | Not e1 => compl (mem_reg e1)
  end.
```

We will later prove that extended regular expressions are equivalent to the inductive definition of regular languages in 3.2.1. In order to do that, we introduce a predicate on regular expressions that distinguishes **standard regular expressions** from **extended regular expressions** (as introduced above). The grammar of standard regular expression is as follows:

$$r, s := \emptyset \mid \varepsilon \mid a \mid r^* \mid r + s \mid rs$$

Note that standard regular expressions are equivalent to regular languages. We realize standard regular expressions as a predicate on extended regular expressions.

```
Fixpoint standard (e: regular_expression char) :=
  match e with
    | Not _ => false
    | And _ _ => false
    | Dot => false
    | _ => true
  end.
```

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 δ.

We define a **run** of a word $w \in \Sigma^*$ on an automaton $A = (Q, s, F, \delta)$ as a sequence of states σ such that for every two consecutive positions $i, i+1$ in σ 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 σ 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.
```

```
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 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 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 δ, we speak of **deterministic** finite automata (DFA). In this case, we write δ as a function in our Coq development.

```
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 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 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).
```

ε-Transitions

Non-deterministic finite automata with ε-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 ε character. They do not “consume” a character of the input and can be taken arbitrarily often. The main disadvantage is that the properties of ε-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 ε-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, δ) fulfills the definition of an NFA. Thus, A *is* an NFA. □

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:*

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

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*

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

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. □

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 \in X -> nfa_accept A x w -> dfa_accept nfa_to_dfa X w.
Lemma nfa_to_dfa_aux1 (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 .
```

4.3 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* $\text{reachable} := \{y \mid (s, y) \in \text{reachable1}^*\}$*, where* reachable1^* *denotes the transitive closure of* reachable1*. With this, we can define the connected automaton A_c:*

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

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 \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 |}.
```

Lemma 4.3.2. *For every state $x \in$ 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 $x \in$ *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 $x \in$ reachable we get $y \in$ 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 $s \in$ 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.

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, $w \notin \mathcal{L}(A_c)$. □

Thus, emptiness is decidable.

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

```
Lemma dfa_lang_empty_correct:
  reflect (dfa_lang A1 =i 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.*

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

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.

```
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 =i 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 Σ. 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: ~~ dfa_accept 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.*

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

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

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 $x \in F \iff x \notin Q \backslash F$. 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. $\mathcal{L}(A_\neg) = \Sigma^* \backslash \mathcal{L}(A)$.*

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 ε-transition. We do not allow ε-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:*

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

```
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 σ be a run starting in x for $w \in \Sigma^*$ on A_2. By induction on σ.

1. For $\sigma = x$ we have $w = \varepsilon$. Therefore, we have that σ is also a run starting in x for ε on A_{Conc}.

2. For $\sigma = xy\sigma'$ we have $w = aw'$, $(x, a, y) \in \delta_2$. By definition of δ_{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
–> nfa_run nfa_conc (inr _ 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 –>
nfa_lang A2 w –>
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 $w_1 w_2 \in \mathcal{L}_x(A_{Conc})$.*

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 –>
nfa_lang A2 w2 –>
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*

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

or

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

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 ε-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 nfa_star := (dfa_disj dfa_eps (nfa_to_dfa nfa_repeat)).
Lemma nfa_star_correct: dfa_lang nfa_star =i star (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. `nfa_star` takes an NFA but returns a DFA, whereas `nfa_conc` expects *and* returns NFAs.

```
Fixpoint re_to_dfa (r: regular_expression char): dfa char :=
  match r with
  | Void => dfa_void char
  | Eps => dfa_eps char
  | Dot => dfa_dot char
  | Atom a => dfa_char char a
  | Star s => nfa_star (dfa_to_nfa (re_to_dfa s))
  | Plus s t => dfa_disj (re_to_dfa s) (re_to_dfa t)
  | And s t => dfa_conj (re_to_dfa s) (re_to_dfa t)
  | Conc s t => nfa_to_dfa (nfa_conc (dfa_to_nfa (re_to_dfa s)) (dfa_to_nfa (re_to_dfa t)))
  | Not s => dfa_compl (re_to_dfa s)
  end.
Lemma re_to_dfa_correct r: dfa_lang (re_to_dfa r) =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 Σ 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 re_equiv r s := dfa_equiv (re_to_dfa r) (re_to_dfa s).
Lemma re_equiv_correct r s: re_equiv r s <-> r =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 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 "**Brzozowski's method**" [13] and builds upon Brzozowski 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.

Brzozowskis 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, δ).

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 σ 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* ε.

In the case of a non-empty X, we make the following observation. If we pick an element $z \in X$, then any run σ 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 σ 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 σ 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 $X\backslash\{z\}$. 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 σ does not pass through z, it is covered by the regular expression for paths from x to y restricted to $X\backslash\{z\}$.

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 w.r.t. to this ordering.

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

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

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. nPlus is $\sum$ on regular expressions. dfa_step_any is the list chars of that contains all transitions from x to y. R0 covers the case of $X = \emptyset$.

```
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.
```

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.*

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

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

Definition 4.8.3. *Let $w \in \Sigma^*$. Let $X \subseteq Q$, and $x, y \in X$. Let σ 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 `belast` to remove the last element from a sequence. Note that, mathematically, runs include the starting state. In our formaliziation, 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.*

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

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 $L^X_{x,y}$ 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 σ be the run of w on A starting in x. Let $z \in \sigma_1 \ldots \sigma_{|\sigma|-1}$. Then there exist $w_1, w_2 \in \Sigma^*$ such that*

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

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

```
Lemma run_split x z w: z \in dfa_run' A x w ->
    exists w1, exists w2,
      w = w1++w2 /\
      size w2 < size w /\
      z \notin belast (dfa_run' A x w1) /\
      last x (dfa_run' A x w1) = 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 σ 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 $X = (X \cup \{z\}) \backslash \{z\}$ 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 L_cat (X: {set A}) x y z w1 w2:
  z \in X ->
  w1 \in L^X x z ->
  w2 \in L^X z y ->
  w1++w2 \in L^X x y.
```

Lemma 4.8.9. *Let $n \in \mathbb{N}$. Let $w_0, \ldots, w_{n-1} \in L^X_{z,z}$. We have that*

$$w_0 \ldots w_{n-1} \in L^{X \cup \{z\}}_{z,z}.$$

Proof. By induction on n.

1. For $n = 0$ we have to prove $\varepsilon \in L^{X\cup\{z\}}_{z,z}$ which holds by Lemma 4.8.5.

2. For $n = n' + 1$ we have $w_0 \ldots w_{n-2} \in L^{X\cup\{z\}}_{z,z}$ by inductive hypothesis. We also have $w_0, \ldots, w_{n-1} \in L^X_{z,z}$ by assumption, and, thus, $w_{n-1} \in L^X_{z,z}$. By Lemma 4.8.8 we get $w_0 \ldots w_{n-1} \in L^{X\cup\{z\}}_{z,z}$.

□

```
Lemma L_flatten (X: {set A}) z vv:  all (L^X z z) vv ->
    flatten vv \in L^(z |: X) z 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*

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

Proof. "⇒" By induction on $|w|$.

1. For $|w| = 0$ we get $w \in L^X_{x,y}$ by Lemma 4.8.5.

2. For $|w| > 0$ we get w_1 and w_2 such that $w = w_1w_2$, $w_1 \in L^X_{x,z}$ and $w_2 \in L^{X\cup\{z\}}_{z,y}$. By inductive hypothesis we get

$$w_2 \in L^X_{z,z}(L^X_{z,z})^* L^X_{z,y} \;\vee\; w_2 \in L^X_{z,y}$$

The latter gives us $w \in L^{X\cup\{z\}}_{x,y}$ by Lemma 4.8.8. With the former, we have w_3, w_4, and w_4 such that $w_2 = w_3w_4w_5$, $w_3 \in L^X_{z,z}$, $w_4 \in (L^X_{z,z})*$ and $w_5 \in L^X_{z,y}$. We merge w_3 and w_4 such that $w_3w_4 \in (L^X_{z,z})^*$. This gives us $w_2 \in (L^X_{z,z})^* L^X_{z,y}$. Thus, $w_1w_2 \in L^X_{x,z}(L^X_{z,z})^* L^X_{z,y}$.

"⇐" We have $w_1 \in L^X_{x,z}$, $w_2 \in (L^X_{z,z})^*$, and $w_3 \in L^X_{z,y}$. By Lemma 4.8.9 we get $w_2 \in L^{X\cup\{z\}}_{z,z}$. Thus, $w_1w_2w_3 \in L^{X\cup\{z\}}_{x,y}$ by Lemma 4.8.8. □

```
Lemma L_rec (X: {set A}) x y z:
  L^(z |: X) x y =i plus (conc (L^X x z) (conc (star (L^X z z)) (L^X z y)))
                         (L^X x y).
```

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

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

$$L^X_{x,y} = R^X_{x,y}.$$

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

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

By Lemma 4.8.10 we also know that

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

The claim follows by inductive hypothesis.

□

```
Lemma L_R n (X: {set A}) x 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 $x, y \in Q$. $R^X_{x,y}$ is a standard regular expression.*

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

```
Lemma R_standard (X: {set A}) x 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. "⇒" Let L be regular. Then, by definition of regular languages, there is an equivalent standard regular expression.

"⇐" 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. □

```
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.

5 Myhill-Nerode

In this chapter, we consider three additional characterizations of regular languages:

1. Myhill relations,
2. weak Nerode relations,
3. and Nerode relations.

We will show that these three characterizations can be used to characterize regular languages by proving them equivalent to the existence of a (deterministic) finite automaton.

5.1 Definitions

Before we can state the Myhill-Nerode theorem, we need a number of auxiliary definitions. We roughly follow [22].

Definition 5.1.1. *Let $\equiv$ be an equivalence relation. The **equivalence class** of $u \in \Sigma^*$ w.r.t. $\equiv$ is the set of all v such that $u \equiv v$. It is denoted by $[u]_\equiv$.*

Definition 5.1.2. *Let $\equiv$ be an equivalence relation. $\equiv$ is of **finite index** if and only if the set of $\{[u]_\equiv \mid u \in \Sigma^*\}$ is finite.*

Due to the lack of native support for quotient types in Coq, we formalize equivalence relations of finite index as surjective functions from Σ^* to a finite type X.

Definition 5.1.3. *Let X be finite. Let $f : \Sigma^* \mapsto X$ be surjective. Let $u, v \in \Sigma^*$. f is an **equivalence relation of finite index**. u and v are equivalent w.r.t. f if and only if $f(u) = f(v)$. $f(u)$ is the equivalence class of u w.r.t. f.*

```
Record Fin_Eq_Cls :=
      { fin_type  : finType;
        fin_func  :> word -> fin_type;
         fin_surjective  :  surjective  fin_func  }.
```

Definition 5.1.4. *Let f be as above. Let $x \in X$. $w \in \Sigma^*$ is a **representative** of x if and only if $f(w) = x$. Since f is surjective, every w has a representative. We write $cr(x)$ to denote the **canonical representative** of x, which we obtain by constructive choice.*

```
Definition cr (f: Fin_Eq_Cls) x := xchoose ( fin_surjective  f x).
```

5.1.1 Myhill Relations

Definition 5.1.5. *Let $\equiv$ be an equivalence relation of finite index. $\equiv$ is a **Myhill relation** [22] for L if and only if*

*(i) $\equiv$ is **right congruent**, i.e. for all $u, v \in \Sigma^*$ and $a \in \Sigma$,*

$$u \equiv v \Rightarrow u \cdot a \equiv v \cdot a.$$

*(ii) $\equiv$ **refines** L, i.e. for all $u, v \in \Sigma^*$,*

$$u \equiv v \Rightarrow (u \in L \iff v \in L).$$

Myhill relations are commonly referred to as "Myhill-Nerode relations". In this thesis, it makes sense to split the concept of a Myhill relation from that of the Nerode relation. Apart from the Nerode relation, which can be seen as the coarsest Myhill relation, we also define weak Nerode relations that have no direction connection to Myhill relations. Thus, we strictly separate the characterizations.

Mathematically, Myhill relations are required to be of finite index. We only formalize equivalence relations of finite index. Thus, proving that a Myhill relation is of finite index mathematically corresponds to constructing a Myhill relation in our formalization.

```
Definition right_congruent {X} (f: word -> X) :=
        forall u v a, f u = f v -> f (rcons u a) = f (rcons v a).
Definition refines {X} (f: word -> X) :=
        forall u v, f u = f v -> u \in L = (v \in L).
Record Myhill_Rel :=
        { myhill_func :> Fin_Eq_Cls;
          myhill_congruent : right_congruent myhill_func;
           myhill_refines : refines myhill_func }.
```

Myhill relations correspond to the equivalence relations defined as the pairs of words (u, v) whose runs on a DFA A end in the same state. These relations are right congruent, refine $\mathcal{L}(A)$ and are of finite index as A has finitely many states. We will later give a formal proof of this.

5.1.2 Nerode Relations

Definition 5.1.6. *Let $u, v \in \Sigma^*$. Let L be a language. We define the **Nerode relation** $\doteq_L$ for L such that*

$$u \doteq_L v \iff \forall w \in \Sigma^*.\ uw \in L \Leftrightarrow vw \in L.$$

The Nerode relation given above is a propositional statement in Coq. To proof that the Nerode relation is of finite index, we require an equivalence relation, i.e. a function f from words to a finite type, such that f is equivalent to $\doteq_L$.

```
Definition equiv_suffix {X} (f: word -> X) :=
    forall u v, f u = f v <-> suffix_equal u v.
Record Nerode_Rel :=
   { nerode_func :> Fin_Eq_Cls;
     nerode_equiv: equiv_suffix nerode_func }.
```

Definition 5.1.7. *Let L be a language and let $\equiv$ be an equivalence relation. We say that $\equiv$ is a **weak Nerode relation** for L if and only if*

$$\forall u, v \in \Sigma^*.\ u \equiv v \implies u \doteq_L v.$$

```
Definition suffix_equal u v :=
    forall w, u++w \in L = (v++w \in L).
Definition imply_suffix {X} (f: word -> X) :=
      forall u v, f u = f v -> suffix_equal u v.
Record Weak_Nerode_Rel :=
     { weak_nerode_func :> Fin_Eq_Cls;
       weak_nerode_imply: imply_suffix weak_nerode_func }.
```

It appears that the notion of a weak Nerode relation is not found in the literature. We will later prove them weaker than Myhill relations, in the sense that every Myhill relation is also a weak Nerode relation.

5.1.3 Myhill-Nerode Theorem

We can now move on to the statement of the Myhill-Nerode theorem [22].

Theorem 5.1.8. *(Myhill-Nerode) Let L be a language. The following four statements are equivalent:*

1. *there exists a deterministic finite automaton that accepts L;*
2. *there exists a Myhill relation for L;*
3. *there exists a weak Nerode relation for L;*
4. *the Nerode relation for L is of finite index.*

Our proof of equivalence will have the following structure:

$$\underbrace{\mathit{DFA} \Longrightarrow \rlap{$\overbrace{\phantom{\mathit{Myhill} \Longrightarrow \mathit{weak\ Nerode}}}^{\text{b}}$}\mathit{Myhill}}_{\text{a}} \Longrightarrow \underbrace{\mathit{weak\ Nerode} \Longrightarrow \rlap{$\overbrace{\phantom{\mathit{Nerode} \Longrightarrow \mathit{DFA}}}^{\text{d}}$}\mathit{Nerode}}_{\text{c}} \Longrightarrow \mathit{DFA}.$$

We will first show (a), (b), and (d). We will then give a proof of (c), which is the most challenging proof and formalization in this chapter.

5.2 Finite Automata to Myhill relations

We assume we are given a DFA A. We will be using the states of A as equivalence classes. Our goal is to construct a Myhill relation, for which we will need an equivalence relation of finite index. Therefore, we first need to ensure that the mapping from words to equivalence classes is surjective. Thus, we consider the equivalent, connected automaton $A_c = (Q_c, s_c, F_c, \delta_c)$ (Definition 4.3.1), which has only reachable states. This enables us to construct a surjective function from words to the states of A_c.

Definition 5.2.1. *Let $u \in \Sigma^*$. Let σ be the run of u on A_c. We define $f_M : \Sigma^* \mapsto Q_c$ such that $f_M(U)$ is the last state in σ, i.e.*

$$f_M(u) := \sigma_{|\sigma|-1}.$$

Note that f_M is surjective (follows directly from Lemma 4.3.5) and, thus, an equivalence relation of finite index.

```
Definition f_M := fun w => last (dfa_s A_c) (dfa_run A_c w).
Lemma f_M_surjective: surjective f_M.
Definition f_fin : Fin_Eq_Cls :=
    {| fin_func := f_M;
       fin_surjective := f_M_surjective |}.
```

In order to show that f_M is a Myhill relation, we prove that it fulfills Definition 5.1.5.

Lemma 5.2.2. *f_M is right congruent.*

Proof. Let $u, v \in \Sigma^*$ such that $f_M(u) = f_M(v)$. Let $a \in \Sigma$. Since A is deterministic, we get $f_M(ua) = f_M(va)$. □

Lemma 5.2.3. *f_M refines $\mathcal{L}(A_c)$.*

Proof. Let $u, v \in \Sigma^*$ such that $f_M(u) = f_M(v)$. By definition of f_M, the runs u and v on A end in the same state. Thus, either u and v are both accepted, or both not accepted. □

Theorem 5.2.4. *f_M is a Myhill relation for $\mathcal{L}(A)$.*

Proof. By Lemma 4.3.3, we have $\mathcal{L}(A_c) = \mathcal{L}(A)$. Thus, it suffices to show that f_M is a Myhill relation for $\mathcal{L}(A_c)$. This follows with Lemma 5.2.2 and Lemma 5.2.3. □

We only have extensional equality on $\mathcal{L}(A_c)$ and $\mathcal{L}(A)$ in Coq. Thus, we first show that f_M is a Myhill relation for $\mathcal{L}(A_c)$. Then, we show that we can get a Myhill relation for $\mathcal{L}(A)$ from a Myhill relation for $\mathcal{L}(A_c)$.

```
Definition dfa_to_myhill' : Myhill_Rel (dfa_lang A_c) :=
    {| myhill_func := f_fin ;
      myhill_congruent := f_M_right_congruent;
       myhill_refines := f_M_refines |}.
Lemma myhill_lang_eq L1 L2: L1 =i L2 -> Myhill_Rel L1 -> Myhill_Rel L2.
Lemma dfa_to_myhill : Myhill_Rel (dfa_lang A).
```

This concludes the proof of step (a).

5.3 Myhill relations to weak Nerode relations

We show that, if there exists a Myhill relation, there also exists a weak Nerode relation. In fact, we will prove that any Myhill relation *is* a weak Nerode relation.

Theorem 5.3.1. *Let f be a Myhill relation for a language L. Then f is a weak Nerode relation for L.*

Proof. Let $u, v \in \Sigma^*$ such that $u =_f v$. We have to show that for all $w \in \Sigma^*$, $uw \in L \Leftrightarrow vw \in L$. By induction on w.

1. For $w = \varepsilon$, we get $u \in L \Leftrightarrow v \in L$ as f refines L.

2. For $w = aw'$, we get $ua =_f va$ by congruence of f and thus, by inductive hypothesis, $uaw' \in L \Leftrightarrow vaw' \in L$.

$\square$

```
Lemma myhill_suffix: imply_suffix L f.
Lemma myhill_to_weak_nerode: Weak_Nerode_Rel L.
  Proof. exact
    {| weak_nerode_func := f;
      weak_nerode_imply := myhill_suffix |}.
  Defined.
```

This concludes step (b) of Theorem 5.1.8.

5.4 Nerode relations to Finite Automata

We prove step (d) of Theorem 5.1.8. If the Nerode relation for a language L is of finite index, we can construct a DFA that accepts L. The construction is very straightforward and uses the equivalence classes of the Nerode relation as the set of states for the automaton.

Definition 5.4.1. *Let L be a language. Let X be a finite type. Let $f : \Sigma^* \mapsto X$ be the equivalence relation which proves that the Nerode relation for L is of finite index. We construct DFA A such that*

$$\begin{aligned} s &:= f(\varepsilon) \\ F &:= \{x | x \in X \wedge cr(x) \in L\} \\ \delta &:= \{(x, a, f(cr(x)a)) \mid x \in X, a \in \Sigma\} \\ A &:= (X, s, F, \delta). \end{aligned}$$

```
Definition nerode_to_dfa :=
  {| dfa_s := f [::];
     dfa_fin := [pred x | cr f x \in L ];
     dfa_step := [fun x a => f (rcons (cr f x) a)] |}.
```

In order to show that A accepts the language L, we first need to connect runs on A to the equivalence classes, i.e. the range of f. The following lemma gives a direct connection.

Lemma 5.4.2. *Let $w \in \Sigma^*$. Let σ be the run of w on A starting in s. We have that the last state of σ is the equivalence class of w, i.e.*

$$\sigma_{|\sigma|-1} = f(w).$$

Proof. We proceed by induction on w from right to left.

1. For $w = \varepsilon$ we have $s = f(\varepsilon)$.

2. For $w = w'a$ we know that the run of w' on A starting in s ends in $f(w')$. It remains to show that $(f(w'), a, f(w)) \in \delta$. We have $cr(f(w'))a =_f w$, i.e. $f(cr(f(w'))a) = f(w)$ by definition of f. Thus, it suffices to show $(f(w'), a, f(cr(f(w'))a)) \in \delta$, which holds by definition of δ.

□

Theorem 5.4.3. *A accepts L, i.e. $\mathcal{L}(A) = L$.*

Proof. Let $w \in \Sigma^*$. Let σ be the run of w on A starting in s. w is accepted if and only if $\sigma_{|\sigma|-1} \in F$, i.e. if and only if $cr(\sigma_{|\sigma|-1}) \in L$. We have $w =_f cr(\sigma_{|\sigma|-1})$ and therefore $w \in L \Leftrightarrow cr(\sigma_{|\sigma|-1}) \in L$. Thus w is accepted if and only if $w \in L$. □

The resulting automaton is minimal, i.e. there exists no other automaton that accepts the same language and has less states than A.

This concludes step d of Theorem 5.1.8.

5.5 Minimizing Equivalence Classes

Finally, we prove that if there is a weak Nerode relation for a language L, the Nerode relation is of finite index. For this purpose, we employ a table-filling algorithm [19] to find indistinguishable states under the Myhill-Nerode relation. This algorithm was originally intended to be used on automata. It identifies (un)distinguishable states based on their successors. We use the finite type X, i.e., the equivalence classes, instead of states.

For the remainder of this section, we assume we are given a language L and a weak Nerode relation f_0.

We employ a fixed-point construction to find equivalence classes that are $\doteq_L$-equivalent. In each step, we add those equivalence classes that are distinguishable based on equivalence classes that were distinguishable in the previous step. The initial set of distinguishable equivalence classes are distinguishable by the inclusion of their canonical representative in L. We denote this initial set $dist_0$.

$$dist_0 := \{(x, y) \in F \times F \mid cr(x) \in L \Leftrightarrow cr(y) \notin L\}.$$

```
Definition distinguishable := [ fun x y => (cr f_0 x) \in L != ((cr f_0 y) \in L) ].
Definition dist0 := [set x | distinguishable x.1 x.2 ].
```

To find more distinguishable equivalence classes, we have to identify equivalence classes that "lead" to distinguishable equivalence classes. In analogy to the minimization procedure on automata, we define successors of equivalence classes with respect to a single character. The intuition is that two states are distinguishable if they are succeeded by a pair of distinguishable states. Conversely, if a pair of states is not distinguishable, then their predecessors can not be distinguished by those states. Thus, two states are undistinguishable if none of their succeeding pairs of states are distinguishable.

Definition 5.5.1. *Let $x, y \in X$ and $a \in \Sigma$. We define $succ_a$ and $psucc_a$. $succ_a(x) := f_0(cr(x) \cdot a)$ and $psucc_a(x, y) := (succ_a(x), succ_a(y))$.*

```
Definition succ := [ fun x a => f_0 ((cr f_0 x) ++ [::a]) ].
Definition psucc := [ fun x y => [ fun a => (succ x a, succ y a) ] ].
```

The fixed-point algorithm tries to extend the set of distinguishable equivalence classes by looking for a pair of equivalence classes that transitions to a pair of distinguishable equivalence classes. Given a set of pairs of equivalence classes *dist*, we define the set of pairs of distinguishable equivalence classes that have successors in *dist* as

$$dist_S(dist) := \{(x, y) \mid \exists a.\, psucc_a(x, y) \in dist\}.$$

```
Definition distS (dist : {set X*X}) :=
    [set (x,y) | x in X, y in X & [ exists a, psucc x y a \in dist ] ].
```

Definition 5.5.2. *Let dist be a subset of $X \times X$. We define one-step-dist such that*

$$one\text{-}step\text{-}dist(dist) := dist_0 \cup dist \cup distinct_S(dist).$$

```
Definition one_step_dist dist := dist0 :|: dist :|: (distS dist ).
```

Lemma 5.5.3. *one-step-dist is monotone and has a fixed-point.*

Proof. Monotonicity follows directly from the monotonicity of $\cup$. The number of sets in $X \times X$ is finite. Therefore, *one-step-dist* has a fixed point. We iterate *one-step-dist* $|X * X| + 1 = |X|^2 + 1$ times on the empty set. Since there can only ever be $|X * X|$ items in the result of *one-step-dist*, we will find the fixed point in no more than $|X * X| + 1$ steps.

Let ***distinct*** be the fixed point of *one-step-dist* and let it be denoted by $\ncong$. We write ***equiv*** for the complement of *distinct* and denote it $\cong$. □

Definition lfp := iter #|T|.+1 F set0.

Definition distinct := lfp one_step_dist .

We now show that $\cong$ is equivalent to the Nerode relation. Formally, this means constructing a function that fulfills our definition of an equivalence relation of finite index. Then, we prove that this equivalence relation is equivalent to the the Nerode relation. First, we will prove that $\cong$ is an equivalence relation. Then, we will prove it equivalent to the Nerode relation in two separate steps, since the two directions require different strategies.

Lemma 5.5.4. *$\cong$ is an equivalence relation.*

Proof. We first state transitivity of $\cong$ in terms of $\ncong$:

$$\forall x, y, z \in X.\ \neg(x \ncong y) \implies \neg(y \ncong z) \implies \neg(x \ncong z). \tag{*}$$

It suffices to show that *distinct* is anti-reflexive, symmetric and fulfills (*). Note that complementary transitivity, anti-reflexivity and symmetry are closed under union. We proceed by fixed-point induction.

1. For $\mathit{one\text{-}step\text{-}dist}(\mathit{dist}) = \emptyset$ we have anti-reflexivity, symmetry and (*) by the properties of $\emptyset$.

2. For $\mathit{one\text{-}step\text{-}dist}(\mathit{dist}) = \mathit{dist}'$ we have *dist* anti-reflexive, symmetric and (*). It remains to show that dist_0 and $\mathit{distinct}_S(\mathit{dist})$ are anti-reflexive, symmetric and fulfill (*).

   dist_0 is anti-reflexive, symmetric and fulfills (*) by definition.

   $\mathit{distinct}_S(\mathit{dist})$ can be seen as an intersection of a symmetric subset of $X \times X$ defined by psucc_a and *dist*, the latter of which is anti-reflexive, symmetric and fulfills (*). Thus, $\mathit{distinct}_S(\mathit{dist})$ is anti-reflexive, symmetric and fulfills (*).

   Therefore, dist' is anti-reflexive, symmetric and fulfills (*).

□

Lemma equiv_refl x: x ~= x.

Lemma equiv_sym x y: x ~= y -> y ~= x.

Lemma equiv_trans x y z: x ~= y -> y ~= z -> x ~= z.

Lemma 5.5.5. *Let $u, v \in \Sigma^*$. $f_0(u) \cong f_0(v) \implies u \doteq_L v$.*

Proof. Let $w \in \Sigma^*$. We then show the contraposition of the claim:

$$uw \in L \not\Leftrightarrow vw \in L \implies f_0(u) \not\cong f_0(v).$$

By induction on w.

1. For $w = \varepsilon$ we have $u \in L \not\Leftrightarrow v \in L$ which gives us $(f_0(u), f_0(v)) \in dist_0$. Thus, $f_0(u) \not\cong f_0(v)$.

2. For $w = aw'$ we have $uaw \in L \not\Leftrightarrow vaw \in L$. We have to show $f_0(u) \not\cong f_0(v)$, i.e. $(f_0(u), f_0(v)) \in distinct$. By definition of *distinct*, it suffices to show $(f_0(u), f_0(v)) \in one\text{-}step\text{-}dist(distinct)$.

   For this, we prove $(f_0(u), f_0(v)) \in distinct_S(distinct)$. By $uaw \in L \not\Leftrightarrow vaw \in L$ we have $(f_0(cr(u)a), f_0(cr(v)a)) \in dist_0$.

   It remains to show that $f_0(cr(u)a) \not\cong f_0(cr(v)a)$ which we get by inductive hypothesis. For this, we need to show that $cr(u)aw \in L \not\Leftrightarrow cr(v)aw \in L$.

   By the properties of f, we get $cr(u)aw \in L \Leftrightarrow uaw \in L$ and $cr(v)aw \in L \Leftrightarrow vaw \in L$. Thus, $cr(u)aw \in L \not\Leftrightarrow cr(v)aw$.

□

Lemma 5.5.6. *Let $u, v \in \Sigma^*$. If $f_0(u) \not\cong f_0(v)$, then u and v are **not** equivalent wr.t. the Nerode relation, i.e. $f_0(u) \not\cong f_0(v) \implies u \not\doteq_L v$.*

Proof. We do a fixed-point induction.

1. For $one\text{-}step\text{-}dist(dist) = \emptyset$ we have $(f_0(u), f_0(v)) \in \emptyset$ and thus a contradiction.

2. For $one\text{-}step\text{-}dist(dist) = dist'$ we have $(f_0(u), f_0(v)) \in dist'$. We do a case distinction on $dist'$.

   a) $(f_0(u), f_0(v)) \in dist_0$. We have $u \in L \not\Leftrightarrow v \in L$. Thus, $u \not\doteq_L v$ as witnessed by $w = \varepsilon$.

   b) $(f_0(u), f_0(v)) \in dist$. By inductive hypothesis, $u \not\doteq_L v$.

   c) $(f_0(u), f_0(v)) \in distinct_S(dist)$. We have $a \in \Sigma$ with $psucc_a(f_0(u), f_0(v))) \in dist$. By inductive hypothesis, we get $cr(u)a \not\doteq_L cr(v)a$ as witnessed by $w \in \Sigma^*$ such that $cr(u)aw \in L \not\Leftrightarrow cr(v)aw \in L$.

      By the properties of f, we get $cr(u)aw \in L \Leftrightarrow uaw \in L$ and $cr(v)aw \in L \Leftrightarrow vaw \in L$. Thus, we have $u \not\doteq_L v$ as witnessed by aw.

□

Corollary 5.5.7. *Let $u, v \in \Sigma^*$. We have that*

$$f_0(u) \cong f_0(v) \iff u \doteq_L v.$$

Proof. Follows immediately with Lemma 5.5.5 and Lemma 5.5.6. □

```
Lemma equiv_suffix_equal u v: u ~=_f_0 v -> suffix_equal L u v.
Lemma distinct_not_suffix_equal u v:
    u ~!=_f_0 v ->
    exists w, u ++ w \in L != (v ++ w \in L).
Lemma equivP u v:
     reflect ( suffix_equal L u v)
             (u ~=_f_0 v).
```

Definition 5.5.8. *Let $w \in \Sigma^*$. We define*

$$f_{min}(w) := \{x \mid x \in X,\ f_0(w) \cong x\}.$$

Note that the range of f_{min} is finite (since X is finite) and does not contain the empty set (due to reflexivity of $\cong$).

Lemma 5.5.9. *f_{min} is surjective.*

Proof. Let $s \in range(f_{min})$. Since $s \neq \emptyset$, there exists $x \in X$ such that $x \in s$. We have $f_0(x) = f_0(cr(x))$ and therefore $f_0(x) \cong f_0(cr(x))$ by reflexivity of $\cong$. Thus, $f_0(cr(x)) = s$ since $f_{min}(x) = f_{min}(cr(x)) = s$. □

Lemma 5.5.10. *For all $u, v \in \Sigma^*$ we we have*

$$f_{min}(u) = f_{min}(v) \iff f_0(u) \cong f_0(v).$$

Proof. "⇒" We have $f_{min}(u) = f_{min}(v)$ and thus $f_0(u) \cong f_0(v)$.

"⇐" We have $f_0(u) \cong f_0(v)$. Let $x \in X$. It suffices to show that $f_0(u) \cong x$ if and only if $f_0(v) \cong x$. This follows with symmetry and transitivity of $\cong$. □

Lemma 5.5.11. *f_{min} is equivalent to the Nerode relation, i.e. f_{min} is surjective and for all $u, v \in \Sigma^*$ we have*

$$f_{min}(u) = f_{min}(v) \iff u \doteq_L v.$$

Proof. We have proven surjectivity in Lemma 5.5.9. By Lemma 5.5.10 we have $f_{min}(u) = f_{min}(v)$ if and only if $f_0(u) \cong f_0(v)$. By corollary 5.5.7 we have $f_0(u) \cong f_0(v)$ if and only if $u \doteq_L v$. Thus, $f_{min}(u) = f_{min}(v)$ if and only if $u \doteq_L v$. □

The formalization of f_{min} is slightly more involved than the mathematical construction. We first need to define the finite type of f_{min}'s range, which we do by enumerating all possible values of f_{min}.

```
Definition equiv_repr x := [set y | x ~= y].
Definition X_min := map equiv_repr (enum (fin_type f_0 )).
Definition f_min w := SeqSub (equiv_repr_mem (f_0 w)).
```

We then prove lemmas 5.5.9, 5.5.10 and Theorem 5.5.11 which are consequential and straight-forward.

```
Lemma f_min_surjective: surjective f_min.
Lemma f_minP u v:
      reflect (f_min u = f_min v)
              (u ~=_f_0 v).
Lemma f_min_correct: equiv_suffix L f_min.
Definition f_min_fin : Fin_Eq_Cls :=
      {| fin_surjective := f_min_surjective |}.
```

We can now state the result of this section.

Theorem 5.5.12. *The Nerode relation is of finite index.*

Proof. This follows directly from Lemma 5.5.4 and Lemma 5.5.11. □

```
Lemma weak_nerode_to_nerode: Nerode_Rel L.
```

This concludes step c of Theorem 5.1.8 and, thus, this chapter.

Remarks

The characterizations presented in here are very compact, mathematically. Interestingly, they also lend themselves very well to formalization. Even with the fixed-point algorithm, this entire chapter is formalized in less than 530 lines of code. This is a very reasonable size, considering that we introduce three different characterizations and prove them all equally expressive to finite automata.

6 Conclusion

We give a short overview of the theorems presented in this thesis and their corresponding statements in the Coq development. We then evaluate our choice of the SSReflect plugin. Finally, we discuss opportunities for future work.

6.1 Results

Theorem 4.2.1 and Theorem 4.2.5 show that deterministic and non-deterministic finite automata are equally expressive. For this, we construct two functions `dfa_to_nfa` and `nfa_to_dfa` to convert between the two characterizations.

```
Lemma dfa_to_nfa_correct (A: dfa): dfa_lang A =i nfa_lang (dfa_to_nfa A).
Lemma nfa_to_dfa_correct (A: nfa): nfa_lang A =i dfa_lang (nfa_to_dfa A).
```

We show in Theorem 4.6.1 that there is an equivalent DFA for every extended regular expression. For this, we construct a function `re_to_dfa` to compute an equivalent DFA from an extended regular expressions.

```
Lemma re_to_dfa_correct (r: regular_expression char) : dfa_lang (re_to_dfa r) =i r.
```

Building on that, we prove the decidability of equivalence of regular expressions in Theorem 4.7.1 with the help of a decision procedure for equivalence of finite automata. We give a function `re_equiv` to decide the equivalence of regular expressions.

```
Lemma re_equiv_correct (r s: regular_expression char): re_equiv r s <-> r =i s.
```

Theorem 4.8.2 shows that we can give an equivalent regular expression for every automaton. We construct a function `dfa_to_re` to compute the regular expression.

```
Lemma dfa_to_re_correct (A: dfa): dfa_lang A =i (dfa_to_re A).
```

Based on this and the results from previous chapters, we also show that extended and standard regular expressions are equally expressive and, thus, that extended regular expressions and regular languages are equally expressive. We give a function `ext_re_to_std_re` which constructs an equivalent standard regular expression for every extended regular expression.

```
Lemma ext_re_to_std_re_standard (r: regular_expression char): standard char (ext_re_to_std_re r).
Lemma ext_re_to_std_re_correct (r: regular_expression char): (ext_re_to_std_re r) =i r.
```

With Theorem 5.2.4, we prove that we can construct a Myhill relation for a language from a DFA for that language.

```
Lemma dfa_to_myhill (A: dfa): Myhill_Rel (dfa_lang A).
```

We prove in Theorem 5.3.1 that every Myhill relations is also a weak Nerode relation.

```
Lemma myhill_to_weak_nerode (L: language char): Myhill_Rel L -> Weak_Nerode_Rel L.
```

Theorem 5.5.12 shows that, if there is a weak Nerode relation, the Nerode relation is of finite index.

```
Lemma weak_nerode_to_nerode (L: language char): Weak_Nerode_Rel L -> Nerode_Rel L.
```

Finally, we prove in Theorem 5.4.1 that, if we are given a Nerode relation of finite index for a language, we can construct a DFA that accepts this language.

```
Lemma nerode_to_dfa_correct (L: language char) (f_N: Nerode_Rel L):
  L =i dfa_lang (nerode_to_dfa f_N).
```

6.2 SSReflect

We make extensive use of SSReflect's features in our development. The formalization of finite automata depends crucially on finite types (and, to a lesser extend, finite sets).

We also employ the reflect paradigm whenever possible. It offers a very convenient way of working with propositional and boolean predicates at the same time. The built-in support for changing from propositional to boolean statements lets us choose the most appropriate representation for the task at hand.

Furthermore, the very extensive library of general purpose lemmas in SSReflect enables us to focus on high-level proofs. The sole exception to this is the allbutlast predicate we need for Theorem 4.8.2. However, even in this case, we can mostly rely on the lemmas for all provided by SSReflect. All we need to do is provide a thin layer between the two predicates.

Additionally, the scripting language offered by SSReflect leads to very concise proof scripts. It succeeds in removing some of the bookkeeping overhead.

There are several disadvantages to SSReflect. One is that it not as widely used as Coq itself. This means that the group of people who can understand the proof scripts is small. However, in some cases, it might be sufficient to explain a small subset of SSReflect in order to give an understandable presentation of the formalized statements.

We also lose practical executability. Specifically, the implementation underlying finite types does not lend itself to computation. Since practical executability is not always a requirement, this restriction may not be relevant to some projects, as is the case with our development.

Based on this considerations, we believe that the use of SSReflect is very beneficial to formalizations that do not require exectuability, especially if there is algorithmic content.

6.3 Future Work

There a several possible extensions to our development. Additionally, there are some topics that are not so much extensions but rather candidates for future formalizations.

6.3.1 ε-Transitions

We have avoided ε-transitions in our formalization. Non-deterministic finite automata with ε-transitions and regular expressions are equally expressive. They are, as we have shown, unnecessary to derive the results proven in this thesis. Nonetheless, it would be very interesting to add them to the list of formalized characterizations of regular languages.

6.3.2 Pumping Lemma

The Pumping Lemma [8] gives a sufficient condition for the non-regularity of a language. It is a well-known part of the theory of regular languages and, thus, a good candidate for an extension to our development.

6.3.3 Regular Grammars

Another characterization of regular languages is given by regular grammars. Regular grammars seem to enjoy less popularity than other characterizations. A formalization of formal grammars in general would also be a good starting point to formalize other parts of the Chomsky Hierarchy [14]. The context-free languages could be a good candidate for a formalization. We speculate that pushdown automata could be formalized similarly to how we formalized finite automata.

6.3.4 ω-regular languages

A possible next step after the formalization of regular languages is a formalization of ω-regular languages. There does not seem much literature on formalizing this topic. Such a development could include all commonly used acceptance criteria on ω-automata. This would also make for a good opportunity to study Coq's co-inductive definitions.

Bibliography

[1] *2nd Annual Symposium on Switching Circuit Theory and Logical Design, Detroit, Michigan, USA, October 17-20, 1961*. IEEE Computer Society, 1961.

[2] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[3] Dean N. Arden. Delayed-logic and finite-state machines. In *SWCT (FOCS)* [1], pages 133–151.

[4] Robert L. Ashenhurst and Susan Graham, editors. *ACM Turing award lectures: the first twenty years: 1966-1985*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1987.

[5] Andrea Asperti. A compact proof of decidability for regular expression equivalence. In Beringer and Felty [9], pages 283–298.

[6] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. Regular expressions, au point. *CoRR*, abs/1010.2604, 2010.

[7] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The matita interactive theorem prover. In Bjørner and Sofronie-Stokkermans [10], pages 64–69.

[8] Yehoshua Bar-Hillel, Micha A. Perles, and Eli Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, (14):143–172, 1961.

[9] Lennart Beringer and Amy P. Felty, editors. *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*. Springer, 2012.

[10] Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors. *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*. Springer, 2011.

[11] Thomas Braibant and Damien Pous. Deciding kleene algebras in coq. *Logical Methods in Computer Science*, 8(1), 2012.

[12] J. A. Brzozowski and E. J. McCluskey. Signal Flow Graph Techniques for Sequential Circuit State Diagrams. *IEEE Transactions on Electronic Computers*, EC-12:67–76, 1963.

[13] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[14] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. `http://www.chomsky.info/articles/195609--.pdf` – last visited 14^{th} January 2009.

[15] Thierry Coquand and Vincent Siles. A decision procedure for regular expression equivalence in type theory. In *CPP*, pages 119–134, 2011.

[16] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[17] Yo-Sub Han and Derick Wood. The generalization of generalized automata: Expression automata. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *Implementation and Application of Automata*, volume 3317 of *Lecture Notes in Computer Science*, pages 156–166. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-30500-2_15.

[18] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[19] D.A. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257(3):161 – 190, 1954.

[20] Wolfram Kahl and Timothy G. Griffin, editors. *Relational and Algebraic Methods in Computer Science - 13th International Conference, RAMiCS 2012, Cambridge, UK, September 17-20, 2012. Proceedings*, volume 7560 of *Lecture Notes in Computer Science*. Springer, 2012.

[21] S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1965.

[22] Dexter Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.

[23] Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Autom. Reasoning*, 49(1):95–106, 2012.

[24] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[25] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[26] Nelma Moreira, David Pereira, and Simão Melo de Sousa. Deciding regular expressions (in-)equivalence in coq. In Kahl and Griffin [20], pages 98–113.

[27] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959.

[28] Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors. *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*. Springer, 2011.

[29] Chunhan Wu, Xingyuan Zhang, and Christian Urban. A formalisation of the myhill-nerode theorem based on regular expressions (proof pearl). In van Eekelen et al. [28], pages 341–356.