Constructive Formalization of Regular Languages

Base revision 51d3101, Tue Sep 25 15:12:18 2012 +0200, Jan-Oliver Kaiser.

Jan-Oliver Kaiser

September 25, 2012

Abstract

Existing formalizations of regular languages in constructive settings are mostly limited to regular expressions and finite automata. Furthermore, these usually require in the order of 10,000 lines of code. The goal of this thesis is to show that an extensive, yet elegant formalization of regular languages can be achieved in constructive type theory. In addition to regular expressions and finite automata, our formalization includes the Myhill-Nerode theorem. The entire development weighs in at approximately 3,300 lines of code.

Citations?

Reduce & update

Contents

Chapter 1

Introduction

Regular languages are a well-studied class of formal languages. In their current form, they were first studied by Kleene [11], who introduced regular expressions. A precursor [**?**] was suggested by earlier by McCulloch and Pitts ([**?**]). Regular languages were used to describe neural activity in nerve nets. The concept of deterministic finite automata was introduced before Kleene's invention of regular expressions by Huffman and Moore ([**?**], [**?**], [**?**]). Rabin and Scott later introduced the concept non-deterministic finite automata [**?**], for which they were given the Turing award [1].

History: Myhill-Nerode, Brzozowski?

Theoretical importance: Second-Order logic, Monoids

Practical importance? Posix

We will prove and formalize the equivalence of differnt characterizations of regular languages: regular expressions, finite automata and the characterizations given by Myhill-Nerode theorem.

1.1 Recent work

There have been many publications on regular languages in recent years. Many of them investigate decidability of equivalence of regular expressions with a focus on automatically deciding Kleene algebras ([6], [2], [13], [4], [16]). There has also been a paper on formalizing the Myhill-Nerode theorem using only regular expressions and not, as is commonly done, finite automata ([18]). The authors state that this unusual choice stems, at least partly, from the restrictions of Isabelle/HOL (and similar HOL-based theorem provers) which effectively prevents straight-forward formalizations of finite automata. This restriction does not apply to Coq. In fact, our formalization of finite automata turns out to be very close to the mathematical definitio turns out to be very close to the mathematical definition.

1.2 Contributions

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. The contribution of this thesis is in the way in which we formalize these well-known results. We focused not on executability but rather on staying close to the mathematical definitions without sacrificing convenience. Our development shows that Coq (with SSReflect) is well suited for this kind of formalization. We have also developed a new characterization derived from the Myhill-Nerode theorem, which we prove equivalent to the other characterizations.

1.3 Outline

In Chapter 2 we introduce the Coq framework and the SSReflect extension. We give a brief introduction of the SSReflect-specific syntax and 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 the equivalence of regular expressions and finite automata.

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 equivalent to finite automata. The formalization of these characterizations is interesting in itself due to the fact that we had to find a suitable representation of quotient types in Coq, which has no notion of quotient types.

Chapter 2

Coq and SSReflect

We decided to employ the Small Scale Reflection Extension (**SSReflect**[1]) for the **Coq**[2] proof assistant. 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

Description, citation

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 [8].

2.2.1 Finite Types and Ordinals

The most important feature of SSReflect for our purpose are finite types. Finite types are based on lists. Every element of a finite type is contained in the associated (de-duplicated) list and vice versa. SSReflect's support for finite types is based on canonical structures, instances of which come predefined for basic finite types and type constructors. This allows us to easily combine basic finite types such as `bool` with type constructors such as `option` and `sum`. We can also create finite types directly from lists.

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

SSReflect provides boolean versions of the universal and existential quantifiers on finite types, **forallb** and **existsb**. We can compute the number of elements in a finite type F with #|F|. enum gives a list of all items of a finite type. Finite types also come with enumeration functions which provide a consistent ordering. The corresponding functions are enum_rank and enum_val. The input of enum_val and the result of enum_rank are ordinals, i.e. values in [0, \#|F|−1]. The corresponding type can be written as I_#|F|

2.2.2 Boolean Reflection

SSReflect offers boolean reflections for decidable propostions. This allows us to switch back and forth between equivalent boolean and propositional predicates.

2.2.3 Boolean Predicates

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 replaced with one of U (union), I (intersection) or D (difference). 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 and return predicates.

Chapter 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 [9]. 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 $w_1 \dots w_k$ 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, take Coquand and Siles's [6] 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 => existsb 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 $w_2 \in L_2$. □

Listing 3.1: Formalization of lemma 3.1.1

```
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 $w_1 \ldots w_l$ 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 mathemitcal definition as it requires all words from L to be nonempty. Mathematically, however, this is no restriction.

Listing 3.2: Formalization of lemma 3.1.2

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

Theorem 3.1.1. *The decidable languages are closed under concatenation, Kleene star, union, intersection and complement.*

Proof. We have already give 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. $\square$

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:*

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

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

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 semantics of these constructors are 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 ([6]), 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.

Listing 3.3: Regular Expressions

```
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 this definition is 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:

Connect standard reg-exp to reg. languages

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

Chapter 4

Finite Automata

Another way of characterizing regular languages are finite automata (FA)[9]. We will show that the languages of finite automata are exactly the regular languages. Furthermore, we will also derive a decision procedure for equivalence of regular expressions.

4.1 Definition

A finite automaton consists of

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

We define a **run** of a word $w \in \Sigma^*$ on an automaton $A = (Q, s, F, \delta)$ as any sequence of states σ such that $\forall\, 0 <= i < |\sigma| - 2. (\sigma_i, w_i, \sigma_{i-1}) \in \delta$. 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 exist multiple distinct runs for a word. This is the case if and only if δ is not functional.

Listing 4.1: Non-Deterministic Finite Automata

```
Record nfa : Type :=
  {
```

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

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

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

4.1.2 Deterministic Finite Automata

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

Listing 4.2: Deterministic Finite Automata

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

Again, we avoid runs in our formalization of the acceptance criterion in favor of a acceptance criterion that is easier to work with in proofs. In this case, however, we can give a function that computes the unique run of a word on A. This allows us to give an alternative acceptance criterion that is closer to the mathematical definition. We also prove that both criteria are equivalent.

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

Equivalence of Automata

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

Equivalence between DFA and NFA

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

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

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

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

Lemma 4.1.1. *For all powerset states X and for all states x with $x \in X$ we have that*

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

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

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

□

Lemma 4.1.2. *For all powerset states X and all words $w \in \mathcal{L}_X(A_{det})$ there exists a state x such that*

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

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

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

□

Theorem 4.1.1. *The powerset automaton A_{det} accepts the same language as A, i.e.*

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

Proof. "$\subseteq$" This follows directly from lemma 4.1.1 with $x = s$ and $X = s_{det}$.

"$\supseteq$" From lemma 4.1.2 with $X = s_{det}$ we get $\mathcal{L}_{s_{det}}(A_{det}) \subseteq \mathcal{L}_s(A)$, which proves the claim. □

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

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

4.2 Connected Components

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

Definition 4.2.1. *Let* $A = (Q, s, F, \delta)$ *be a DFA. We define* reachable1 *such that for all* x *and* y, $(x, y) \in \text{reachable1} \iff \exists a, (x, a, y) \in \delta$. *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*:*

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

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

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

Make this understandable

Lemma 4.2.1. *For every state* $x \in \text{reachable}$ *we have that*

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

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

- For $w = \varepsilon$ we have $\varepsilon \in \mathcal{L}_x(A)$ and therefore $x \in F$. With $x \in reachable$ we get $x \in F_c$. Thus, $\varepsilon \in \mathcal{L}_x(A_c)$.

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

□

Theorem 4.2.1. *The language of the connected automaton A_c is identical to that of the original automaton A, i.e.*

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

Proof. By reflexivity, we have $s \in$ reachable. We use lemma 4.2.1 with $x = s$ to prove the claim. □

The formalization of lemma 4.2.1 and theorem 4.2.1 is straight-forward.

```
Lemma dfa_connected_correct' x (Hx: x \in reachable) :
    dfa_accept dfa_connected {|ssvalP := Hx|} =1 dfa_accept A1 x.
Lemma dfa_connected_correct: dfa_lang dfa_connected =1 dfa_lang A1.
```

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

Definition 4.2.2. *A* ***representative*** *of a state x is a word w such that the unique run of w on A_c ends in x.*

Lemma 4.2.2. *We can give a representative for every state $x \in Q_c$.*

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

The formalization of theorem 4.2.2 includes a more general version of the theorem, which facilitates the proof by induction over the path.

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

4.3 Emptiness

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

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

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

Proof. By theorem 4.2.1 we have $\mathcal{L}(A) = \mathcal{L}(A_c)$. Therefore, it suffices to show

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

"$\Leftarrow$" We have $\mathcal{L}(A_c) = \emptyset$ and have to show that for all $x \in Q_c$, $x \notin F_c$. Let $x \in Q_c$. By lemma 4.2.2 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 $x \notin F_c$. "$\Rightarrow$" We have $F_c = \emptyset$ and have to show that for all words w, $w \notin \mathcal{L}(A_c)$. We use $F_c = \emptyset$ to show that the last state of the run of w on A_c is non-final. Thus, $w \notin \mathcal{L}(A_c)$.

Thus, emptiness is decidable. □

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

```
Definition dfa_lang_empty := #|dfa_fin dfa_connected| == 0.
Lemma dfa_lang_empty_complete: dfa_lang dfa_connected =1 pred0 -> dfa_lang_empty.
Lemma dfa_lang_empty_sound: dfa_lang_empty -> dfa_lang dfa_connected =1 pred0.
Lemma dfa_lang_empty_correct:
    reflect (dfa_lang A1 =1 pred0)
            dfa_lang_empty.
```

4.4 Deciding Equivalence of Finite Automata

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

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

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

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

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. □

Listing 4.3: Formalization of theorem 4.4.1

```
Definition dfa_sym_diff :=
   dfa_disj (dfa_conj A1 (dfa_compl A2)) (dfa_conj A2 (dfa_compl A1)).
Lemma dfa_sym_diff_correct:
   dfa_lang_empty dfa_sym_diff <-> dfa_lang A1 =1 dfa_lang A2.
```

4.5 Regular Expressions and Finite Automata

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

4.5.1 Regular Expressions to Finite Automata

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

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

Void

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

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

Lemma 4.5.1. *The language of the empty DFA is empty, i.e.*

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

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

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

Eps

Definition 4.5.2. *We define an automaton that accepts only the empty word, i.e.*

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

Lemma 4.5.2. *A_ε accepts no word in state f, i.e. for all w,*

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

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

Lemma 4.5.3. *The language of A_ε is exactly the singleton set containing the empty word, i.e.*

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

Proof. Let $w \in \Sigma^*$. We do an induction on w. For $w = \varepsilon$ we have $\varepsilon \in \mathcal{L}(A_\varepsilon)$ and $\varepsilon \in \{\varepsilon\}$. Therefore, both directions are trivial. For $w = aw'$ we consider both directions independently.
"$\Rightarrow$" We have $(t, a, f) \in \delta_\varepsilon$ and $w' \in \mathcal{L}_f(A_\varepsilon)$. By lemma 4.5.2, this is a contradiction.
"$\Leftarrow$" We get a straigt-forward contradiction from $aw' \in \{\varepsilon\}$. □

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

reflect?

Dot

Definition 4.5.3. *We define an automaton that accepts the set of all singleton words, i.e.*

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

Lemma 4.5.4. *A_{Dot} does not accept any word in state f, i.e. $\mathcal{L}_f(A_{Dot}) = \emptyset$.*

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

Lemma 4.5.5. *A_{Dot} accepts exactly the empty word in state t, i.e. $\mathcal{L}_t(A_{Dot}) = \{\varepsilon\}$.*

Proof. Let $w \in \Sigma^*$. We do a case distinction on w. For $w = \varepsilon$ we have $\varepsilon in \mathcal{L}_t(A_{Dot})$ by $t \in F_{Dot}$. We also have $\varepsilon \in \{\varepsilon\}$. For $w = aw'$ we get $(t, a, f) \in \delta_{Dot}$. Since $aw' \notin \{\varepsilon\}$ it suffices to show that $w' \notin \mathcal{L}_f(A_{Dot})$, which we have by lemma 4.5.4. □

```
Definition dfa_dot :=
  {| dfa_s := None;
     dfa_fin := pred1 (Some true);
     dfa_step := [fun x b => if x == None then Some true else Some false ] |}.
Lemma dfa_dot_correct'' w: ~~ dfa_accept dfa_dot (Some false) w.
Lemma dfa_dot_correct' w: dfa_accept dfa_dot (Some true) w = (w == [::]).
Lemma dfa_dot_correct w: dfa_lang dfa_dot w = (size w == 1).
```

reflect?

Not

Definition 4.5.4. *Given DFA $A = (Q, s, F, \delta)$, the complement automaton $A_\neg$ is constructed by switching accepting and non-accepting states, i.e.*

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

Lemma 4.5.6. *For every state $x \in Q$, we have that $w \in \Sigma^*$ is accepted in x by $A_\neg$ if and only if it is not accepted in x by A, i.e. $\mathcal{L}_x(A_\neg) = \Sigma^* \backslash \mathcal{L}_x(A)$*

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

Lemma 4.5.7. *$A_\neg$ accepts the complement language of A, i.e. $\mathcal{L}(A_\neg) = \Sigma^* \backslash \mathcal{L}(A)$.*

Proof. This follows directly from lemma 4.5.6 with $x = s$. □

Plus

Definition 4.5.5. *Given DFAs $A_1 = (Q_1, s_1, F_1, \delta_1)$ and $A_2 = (Q_2, s_2, F_2, \delta_2)$ we construct the disjunction automaton in the following way:*

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

Lemma 4.5.8. *For every state* $(x_1, x_2) \in Q_\vee$*, we have that*

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

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

Lemma 4.5.9. $\mathcal{L}(A_\vee) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.

Proof. This follows directly from lemma 4.5.8 with $x = (s_1, s_2)$. □

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

And

Definition 4.5.6. *Given DFAs* $A_1 = (Q_1, s_1, F_1, \delta_1)$ *and* $A_2 = (Q_2, s_2, F_2, \delta_2)$ *we construct the conjunction automaton in the following way:*

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

Lemma 4.5.10. *For every state* $(x_1, x_2) \in Q_\wedge$*, we have that*

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

Proof. This proof is very similar to lemma 4.5.8. We do a proof by induction on $w \in \Sigma^*$. For $w = \varepsilon$ we have, by definition of $F_\wedge$, $\varepsilon \in \mathcal{L}_{(x_1,x_2)}(A_\wedge) \iff \varepsilon \in \mathcal{L}_{x_1}(A_1) \wedge \varepsilon \in \mathcal{L}_{x_2}(A_2)$.
For $w = aw'$ we get $(x_1, a, y_1) \in \delta_1$ and $(x_2, a, y_2) \in \delta_2$. By induction hypothesis, we also have $w' \in \mathcal{L}_{x_1}(A_1) \wedge w' \in \mathcal{L}_{x_2}(A_2)$. Thus, we get $aw' \in \mathcal{L}_{y_1}(A_1) \wedge aw' \in \mathcal{L}_{x_2}(A_2)$. □

Lemma 4.5.11. $\mathcal{L}(A_\wedge) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$.

Proof. This follows directly from lemma 4.5.10 with $x = (s_1, s_2)$. □

```
Definition dfa_conj :=
 {| dfa_s := (dfa_s A1, dfa_s A2);
    dfa_fin := (fun x => dfa_fin A1 x.1 && dfa_fin A2 x.2);
    dfa_step := [fun x a => (dfa_step A1 x.1 a, dfa_step A2 x.2 a)] |}.
Lemma dfa_conj_correct' w x1 x2 :
  dfa_accept A1 x1 w && dfa_accept A2 x2 w
  = dfa_accept dfa_conj (x1, x2) w.
Lemma dfa_conj_correct w:
  dfa_lang A1 w && dfa_lang A2 w
  = dfa_lang dfa_conj w.
```

Conc

Definition 4.5.7. *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:*

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

Case dist. on $s_2 \in F_2$

Lemma 4.5.12. *Every run of A_2 can be mapped to a run in A_3.*

symbol for empty run

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

Include all proofs

4.5.2 Deciding Equivalence of Regular Expressions

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

4.5.3 Finite Automata to Regular Expressions

We prove that there is an equivalent standard regular expression for every finite automaton.

Since we are given an automaton it is not obvious how to partition our proof obligations into smaller parts. We use Kleene's original proof, the transitive closure method. This method recursively constructs a regular expression that is equivalent to the given automaton. Given a DFA A, we first assign some ordering to its states. We then define $R^k_{i,j}$ such that $\mathcal{L}(R^k_{i,j})$ is the set of all words that have a run on A starting in state i that ends in state j without ever leaving a state smaller than k. The base case $R^0_{i,j}$ is the set of all singleton words that are edges between state i and j, and ε if $i = j$. Given $R^k_{i,j}$ we can easily define $R^{k+1}_{i,j}$ based on the observation that only one new state has to be considered:

Insert complete formal definition

$$R^{k+1}_{i,j} = R^k_{i,k} \cdot (R^k_{k,k})^* \cdot R^k_{k,j} + R^k_{i,j}.$$

We make use of SSReflect's ordinals to get an ordering on states. We chose to employ ordinals for i and j, but not for k. This simplifies the inductive definitions on k. It does, however, lead to explicit conversions when k is used in place of i or j. In fact, i and j are states in our Coq implementation. We only rely on ordinals for comparison to k.

Add implementation of R

Furthermore, we define $L^k_{i,j} \subseteq \mathcal{L}(A)$ in terms of runs on the automaton. The relation of $L^k_{i,j}$ to $\mathcal{L}(A)$ can be proven very easily. We will also prove it equivalent to $R^k_{i,j}$. This allows us to connect $R^k_{i,j}$ to $\mathcal{L}(A)$.

```
Definition allbutlast p : pred (seq X) :=
  fun xs => all p (belast xs).
Definition L :=
    [fun k: nat =>
      [fun x y: A =>
        [pred w |
         (last x (dfa_run' A x w) == y)
           && allbutlast (<_k) (dfa_run' A x w)
        ]
      ]
    ].
```

Theorem 4.5.1. *We can express $\mathcal{L}(A)$ in terms of L. L is equivalent to R.*

$$\mathcal{L}(A) = \bigcup_{f \in F} L^{|Q|}_{s,f} = \mathcal{L}(\sum_{f \in F} R^{|Q|}_{s,f}).$$

Proof. By definition, every $w \in \mathcal{L}(A)$ has a run that ends in some $f \in F$. Then, by definition, $w \in L^{|Q|}_{s,f}$.

It remains to show that $\mathcal{L}(R^k_{i,j}) = L^k_{i.j}$. This claim can be proven by induction over k. We begin with the inclusion of $\mathcal{L}(R^k_{i,j})$ in $L^k_{i,j}$. For $k = 0$, we do a case distinction on $i == j$ and unfold R. The resulting three cases ($i == j \land w = \varepsilon$, $i == j \land |w| = 1$, $i <> j \land |w| = 1$) are easily closed.

The inductive step has two cases: A triple concatenation and a simple recursion. The second case is solved by the inductive hypothesis. In the firs case, we split up the concatenation such that

$$w = w_1 \cdot w_2 \cdot w_3 \land w_1 \in \mathcal{L}(R^k_{i,k}) \land w_2 \in \mathcal{L}((R^k_{k,k})^*) \land w_3 \in \mathcal{L}(R^k_{k,j}).$$

The induction hypothesis is applied to w_1 and w_3 to get $w_1 \in L^k_{i,k}$ and $w_3 \in L^k_{k,j}$. We use a lemma by Coquand and Siles that splits w_2 into a sequence of words from $\mathcal{L}(R^k_{k,k})$ to which we can apply the induction hypothesis. Two concatenation lemmas for L are used to merge the sequence of words proven to be in $L^k_{k,k}$, w_1 and w_3. This shows $\mathcal{L}(R^k_{i,j}) \subseteq L^k_{i,j}$.

Next, we show the inclusion of $L^k_{i,j}$ in $\mathcal{L}(R^k_{i,j})$, again by induction over k. The base case is solved by case distinction on $i == j$. The inductive step requires a **splitting lemma** for L which shows that every non-empty word in $L^{k+1}_{i,j}$ is either in $L^k_{i,j}$ or has a non-empty prefix in $L^k_{i,k}$ and a corresponding suffix in $L^{k+1}_{k,j}$. The In the first case, we can apply the induction hypothesis. In the second case, we use size induction on the word, apply the original induction hypothesis to the prefix and the size induction hypothesis to the suffix. We use two concatenation lemmas for R to merge the sub-expression. This finishes the proof. □

Formalizing theorem 4.5.1 requires infrastructure to deal with *allbutlast*. Once this is in place, we can formalize the concatenation lemmas for R and L. These are required later to connect sub-results.

```
Lemma R_catL k i j w1 w2:
    w1 \in R^k i  (k_ord k) ->
    w2 \in R^k.+1 (k_ord k) j ->
    w1++w2 \in R^k.+1 i j.

Lemma L_catL k i j w1 w2:
    w1 \in L^k i (enum_val (k_ord k)) ->
    w2 \in L^k.+1 (enum_val (k_ord k)) j ->
    w1++w2 \in L^k.+1 i j.

Lemma L_catL k i j w1 w2:
    w1 \in L^k i (enum_val (k_ord k)) ->
    w2 \in L^k.+1 (enum_val (k_ord k)) j ->
    w1++w2 \in L^k.+1 i j.
```

We also need the splitting lemma mentioned earlier. This is quite intricate. We could split right after the first character and thereby simplify the lemma. However, the current form has the advantage of requiring simple concatenation lemmas.

```
Lemma L_split k' i j a w:
    let k := k_ord k' in
    (a :: w) \in L^k'.+1 i j ->
    (a :: w) \in L^k' i j \/
    exists w1, exists w2,
      a :: w = w1 ++ w2 /\
      w1 != [::] /\
      w1 \in L^k' i (enum_val k) /\
      w2 \in L^k'.+1 (enum_val k) j.
```

These lemmas suffice to show the claim of theorem 4.5.1.

```
Lemma R_L_star k vv:
  ( forall (i j : 'I_#|A|) (w : word char),
      w \in R^k i j -> w \in L^k (enum_val i) (enum_val j)) ->
   all [predD mem_reg (R^k (k_ord k) (k_ord k)) &
        eps (symbol:=char)] vv ->
   flatten vv \in L^k.+1 (enum_val (k_ord k)) (enum_val (k_ord k)).

Lemma R_L k i j w: w \in R^k i j -> w \in L^k (enum_val i) (enum_val j).

Lemma L_R_1 k i j w:
     ( forall (i j : 'I_#|A|) (w : automata.word char),
      w \in L^k (enum_val i) (enum_val j) -> w \in R^k i j) ->
      w \in L^k.+1 (enum_val i) (enum_val j) -> w \in R^k.+1 i j.

Lemma L_R k i j w: w \in L^k (enum_val i) (enum_val j) -> w \in R^k i j.
```

Fix this mess

Chapter 5

Myhill-Nerode

The last characterization of regular languages that we consider is given by the Myhill-Nerode theorem.

5.1 Definition

The following definitions (taken from [12]) will lead us to the statement of the Myhill-Nerode theorem.

Let $\equiv$ be an equivalence relation on Σ^*. Let L be a language over Σ.

Definition 5.1.1. *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.

(i) $\equiv$ is ***right congruent*** *if and only if 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 if and only if forall $u, v \in \Sigma^*$,*

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

(iii) $\equiv$ is of ***finite index*** *if and only if it has finitely many equivalence classes, i.e.*

$$\{[u]_\equiv \mid u \in \Sigma^*\} \text{ is finite}$$

Definition 5.1.3. *A relation is Myhill-Nerode if and only if it satisfies properties (i), (ii) and (iii).*

Fix everything below this line

Definition 5.1.4. *Given a language L, the coarsest Myhill-Nerode relation $\equiv_L$ is the Myhill-Nerode relation that subsumes every other Myhill-Nerode relation, i.e.*

$$\forall \equiv . \equiv \mathrm{Myhill-Nerode} -> \forall u, v.\, u \equiv v \Rightarrow u \equiv_L v.$$

Listing 5.1: Myhill-Nerode relation

```
Definition MN w1 w2 := forall w3, w1++w3 \in L == (w2++w3 \in L).
```

Theorem 5.1.1. *Myhill-Nerode Theorem. A language L is regular if and only if $\equiv_L$ is of finite index.*

5.2 Finite Partitionings and Equivalence Classes

Coq does not have quotient types. We pair up functions and proofs of certain properties to emulate quotient types.

A finite partitioning is a function from Σ^* to some finite type F. We use this concept to model equivalent classes in Coq. A finite partitioning of the Myhill-Nerode relation is a finite partitioning f that also respects the Myhill-Nerode relation, i.e.,

$$\forall u, v \in \Sigma^*.\, f(u) = f(v) \Leftrightarrow u \equiv_L v.$$

Listing 5.2: Finite partitioning of the Myhill-Nerode relation

```
Definition MN_rel (f: Fin_eq_cls) := forall w1 w2, f w1 == f w2 <-> MN w1 w2.
```

Theorem 5.2.1. *$\equiv_L$ is of finite index if and only if there exists a finite partitioning of the Myhill-Nerode relation.*

Proof. If $\equiv_L$ is of finite index, we use the set equivalence classes as a finite type and construct f such that

$$\forall w.\, f(w) = [w]_{\equiv}.$$

f is a finite partitioning of the Myhill-Nerode relation by definition.

Conversely, if we have a finite partitioning of the Myhill-Nerode relation, we can easily see that $\equiv_L$ must be of finite index since f's values directly correspond to equivalence classes. The image of f is finite. Therefore, $\equiv_L$ is of finite index. □

A more general concept is that of a refining finite partitioning of the Myhill-Nerode relation:

$$\forall u, v \in \Sigma^*.\, f(u) = f(v) \Rightarrow u \equiv_L v.$$

Listing 5.3: Refining finite partitioning of the Myhill-Nerode relation

```
Definition MN_ref (f: Fin_eq_cls ) := forall w1 w2, f w1 == f w2 -> MN w1 w2.
```

We require all partitionings to be surjective. Therefore, every equivalence class x has at least one class representative which we denote $cr(x)$. Mathematically, this is not a restriction since there are no empty equivalence classes. In our constructive setting we would have to give a procedure that builds a minimal finite type F' from F and a corresponding function f' from Σ^* to F' such that f' is surjective and extensionally equal to f.

5.3 Minimizing Equivalence Classes

We will prove that refining finite partitionings can be converted into finite partitionings. For this purpose, we employ the table-filling algorithm to find indistinguishable states under the Myhill-Nerode relation ([9]). However, we do not rely on an automaton. In fact, we use the finite type F, i.e., the equivalence classes, instead of states.

Given a refining finite partitioning f, we construct a fixed-point algorithm. The algorithm initially outputs the set of equivalence classes that are distinguishable by the inclusion of their class representative in L. We denote this initial set $dist_0$.

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

To find more distinguishable equivalence classes, we have to identify equivalence classes that lead to distinguishable equivalence classes.

Definition 5.3.1. *We say that a pair of equivalence classes (x, y) **transitions** to (x', y') with a if and only if*

$$f(cr(x) \cdot a) = x' \wedge f(cr(y) \cdot a) = y'.$$

We denote (x', y') by $ext_a(x, y)$.

The fixed-point algorithm tries to extend the set of distinguishable equivalence classes by looking for a so-far undistinguishable pair of equivalence classes that transitions to a pair of distinguishable equivalence classes.

Definition 5.3.2.

$$unnamed(dist) := dist_0 \cup dist \cup \{(x, y) \mid \exists a.\, ext_a(x, y) \in dist\}$$

Lemma 5.3.1. *unnamed is monotone and has a fixed-point.*

Proof. Monotonicity follows directly from the monotonicity of $\cup$. The number of sets in $F \times F$ is finite. Therefore, *unnamed* has a fixed point. □

Let ***distinct*** be the fixed point of *unnamed*. Let ***equiv*** be the complement of *distinct*.

Finish construction

Theorem 5.3.1. *$f_m in$ is a finite partitioning of the Myhill-Nerode relation on L.*

Add formalization

5.4 Finite Automata and Myhill-Nerode

We prove theorem 5.1.1 by proving it equivalent to the existence of an automaton that accepts L.

5.4.1 Finite Automata to Myhill-Nerode

Given DFA A, for all words w we define $f(w)$ to be the last state of the run of w on A.

Lemma 5.4.1. *f is a refining finite partitioning of the Myhill-Nerode relation on $\mathcal{L}(A)$.*

Proof. The set of states of A is finite. For all u, v and w we that if $f(u) = f(v) = x$, i.e., the runs of u and v on A end in the exact same state x. From this, we get that for all w, runs of $u \cdot w$ and $v \cdot w$ on A also end in the same state. Therefore, $u \cdot w \in \mathcal{L}(A)$ if and only if $v \cdot w \in \mathcal{L}(A)$. □

Theorem 5.4.1. *If L is accepted by DFA A, then there exists a finite partitioning of the Myhill-Nerode relation on L.*

Proof. From lemma 5.4.1 we get a refining finite partitioning f of the Myhill-Nerode relation on $\mathcal{L}(A)$. Since L is accepted by A, $L = \mathcal{L}(A)$. Therefore, f is a refining finite partitioning of the Myhill-Nerode relation on L. By theorem 5.3.1 there also exists a finite partition of the Myhill-Nerode relation on L. □

5.4.2 Myhill-Nerode to Finite Automata

Chapter 6

Conclusion

Bibliography

[1] 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.

[2] Andrea Asperti. A compact proof of decidability for regular expression equivalence. In Beringer and Felty [3], pages 283–298.

[3] 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.

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

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

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

[7] Ding-Shu Du and Ker-I Ko. *Problem Solving in Automata, Languages, and Complexity.* 2001.

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

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

[10] 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.

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

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

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

[14] Peter Linz. *An introduction to formal languages and automata (4. ed.).* Jones and Bartlett Publishers, 2006.

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

[16] Nelma Moreira, David Pereira, and Simão Melo de Sousa. Deciding regular expressions (in-)equivalence in coq. In Kahl and Griffin [10], pages 98–113.

[17] Anil Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, ao 1958.

[18] 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.

[19] 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. [18], pages 341–356.