Constructive Formalization of Regular Languages

Jan-Oliver Kaiser

September 5, 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

Chapter 1

Introduction

Regular languages are a well-studied class of formal languages. We will prove the equivalence of three well-known characterizations of regular languages: regular expressions, finite automata and the characterization given by Myhill-Nerode theorem.

History

Theoretical importance

Practical importance?

1.1 Recent work

There have been many publications on regular languages in recent years. Many of them investigate decidability of equivalence of regular languages, though there have also been new equivalence proofs regarding different characterizations of regular languages.

Chapter 2

Coqand SSReflect

We decided to employ the Small Scale Reflection Extension (**SSReflect**) for the **Coq** 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

2.2 SSReflect

2.2.1 Finite Types and Ordinals

The most important feature of SSReflect for our purpose are finite types. 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 nota-

tion, 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

3.1 Definition

We closely follow the definitions from [4]. 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 $w_1 = a_1 a_2 \cdots a_n$ and $w_2 = b_1 b_2 \cdots b_m$, the **concatenation** of w_1 and w_2 is defined as $a_1 a_2 \cdots a_n b_1 b_2 \cdots b_m$ 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 the 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 Operation on languages

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 Star** (also called Kleene closure) of a language L is denoted L^* and is defined as the set of words $w = w_1 w_2 \cdots w_k$ such that $w_1, w_2, \cdots w_k$ are in L. ε is contained in L^* ($k = 0$). We define the **complement** of a language L as $L \backslash \Sigma^*$, which we denote $\neg L$. Furthermore, we make use of the standard set operations **union** and **intersection**.

We take Coquand and Siles's [2] 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 [::].
```

Lemma 3.1.1. *Let $L_1, L_2, w = a_1 a_2 \cdots a_k$ 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. "⇒" 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.
"⇐" 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).
```

Lemma 3.1.2. *Let $L, w = a_1 a_2 \cdots a_k$ be given. We have that*

$$w \in L^* \iff a_2 \cdots a_k \in res_{a_1}(L) \cdot L^* \vee w = \varepsilon.$$

Proof. "⇒" 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 $w = w_1 w_2 \cdots w_l$ such that $w_1, w_2 \cdots w_l$ are in L. There exists a minimal n such that $|w_n| > 0$ and for all $m < n$, $|w_m| = 0$. Let $w_n = b_1 b_2 \cdots b_p$. 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 $w = a_1 a_2 \cdots a_k = w_n \cdots w_l$. 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 $a_n \cdots a_k = w_1 w_2 \cdots w_l$ such that $w_1, w_2 \cdots w_l$ 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, $w \in L^*$.

□

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 given algorithms for every operator. It remains to show that they are correct. For concatenation and the Kleene star, we have shown in lemma 3.1.1 and 3.1.2 that the formalization is equivalent to the formal definition. The remaining operators are applied directly to the 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:*

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

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

3.2.1 Regular Expressions

Regular expressions mirror the definition of regular languages very closely. 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:

1. $\mathcal{L}(\emptyset) = \emptyset$
2. $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
3. $\mathcal{L}(.) = \Sigma$
4. $\mathcal{L}(a) = \{a\}$
5. $\mathcal{L}(r^*) = \mathcal{L}(r)^*$
6. $\mathcal{L}(r + s) = \mathcal{L}(r) \cup \mathcal{L}(s)$
7. $\mathcal{L}(r\&s) = \mathcal{L}(r) \cap \mathcal{L}(s)$
8. $\mathcal{L}(rs) = \mathcal{L}(r) \cdot \mathcal{L}(s)$

We take the implementation from Coquand and Siles's development ([2]), which is also based on SSReflect and comes with helpful infrastructure for our proofs.

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:

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

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

Connect standard reg-exp to reg. languages

3.2.2 Deciding Language Membership

We make use of **derivatives of regular expressions** ([1]) to decide if a word $w \in \Sigma^*$ is contained in the language $\mathcal{L}(r)$ of the regular expression r. Derivatives are themselves regular expressions and are computed with respect to a single input character. In order to define derivatives, we first define a related concept.

Definition 3.2.2. *The derivative der a r of r w.r.t. to a is defined such that*

$$\forall w \in \Sigma^*. w \in \mathcal{L}(der\, a\, r) \Leftrightarrow w \in residual\, a\, \mathcal{L}(r).$$

A suitable implementation is provided by Coquand and Siles.

Listing 3.4: Derivatives of Regular Expressions

```
Fixpoint der x e :=
  match e with
  | Void => Void
  | Eps => Void
  | Dot => Eps
  | Atom y => if x == y then Eps else Void
  | Star e1 => Conc (der x e1) (Star e1)
  | Plus e1 e2 => Plus (der x e1) (der x e2)
  | And e1 e2 => And (der x e1) (der x e2)
```

```
| Conc e1 e2 => if has_eps e1 then
    Plus (Conc (der x e1) e2) (der x e2)
    else Conc (der x e1) e2
| Not e1 => Not (der x e1)
end.
```

Theorem 3.2.1. *For all r, w and a, we have that $w \in der\, a\, r$ if and only if $w \in residual\, a$.*

Proof. We prove the claim by induction over r. Two cases are non-trivial: □ Proof

Given the defining property of derivatives, we can easily see that a generalization of *der* to words suffices to decide language membership. We only need to check if the derivative w.r.t. to a given word accepts the empty word.

```
Fixpoint mem_der e u := if u is x :: v then mem_der (der x e) v else has_eps e.
```

Theorem 3.2.2. *The language of a regular expression r is decidable, i.e.*

$$w \in \mathcal{L}(r) \Leftrightarrow \varepsilon \in \mathcal{L}(mem\_der\, r\, w).$$

Proof. . □ Proof

Chapter 4

Finite Automata

Another way of characterizing regular languages are finite automata. We will show that the languages of finite automata are exactly REG. 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. an alphabet Σ,
3. a starting state $s_0 \in Q$,
4. a set of final states $F \subseteq Q$
5. and a state-transition relation δ. [4]

We define a **run** of a word $w \in \Sigma^*$ on an automaton $A = (\Sigma, Q, s_0, F, \delta)$ 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 if and only if there exists a run σ of w on A such that $\sigma_0 = s_0 \wedge \sigma_{|\sigma|-1} \in F$. The **language** of A is exactly the set of words accepted by A and is denoted $\mathcal{L}(A)$. It will later be useful to also have a acceptance criterion defined by runs starting in a given state $x \in Q$, for which we will denote the resulting language $\mathcal{L}_x(A)$.

Introduce section-wide variables

4.1.1 Determinism and Non-Determinism

Finite automata can be non-deterministic 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 :=
  nfaI {
    nfa_state :> finType;
    nfa_s0 : nfa_state ;
    nfa_fin : pred nfa_state ;
    nfa_step : nfa_state -> char -> pred nfa_state
    }.

Fixpoint nfa_lpath x (xs : seq A) (w: word) {struct xs} :=
match xs,w with
  | y :: xs', a::w' => nfa_step A x a y && nfa_lpath y xs' w'
  | [::] , [::] => true
  | _ , _ => false
end.
```

For functional δ, we speak of **deterministic finite automata**. In this case, we also assume δ to be total and write it as a function. This allows us to directly define the acceptance criterion in terms of the unique run of a word on the automaton.

Listing 4.2: Deterministic Finite Automata

```
Record dfa : Type :=
  dfaI {
    dfa_state :> finType;
    dfa_s0 : dfa_state ;
    dfa_fin : pred dfa_state ;
    dfa_step : dfa_state -> char -> dfa_state
    }.

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.
```

Equivalence between DFA and NFA

Deterministic and non-deterministic finite automata are equally powerful. One direction is trivial since every DFA is also 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: The new set of states is the powerset of the given NFA's set of states. The new starting state is the singleton set containing the original starting state. A state is final if and

only if it contains an original final state. The transition function on powerset states is defined as follows:

$$(P, a, Q) \in \delta_{det} \iff Q = \bigcup_{p \in P} \{q | (p, a, q) \in \delta\}.$$

Listing 4.3: Powerset Construction

```
Definition powerset_state : finType := [finType of {set A}].
Definition powerset_s0 : powerset_state := set1 (nfa_s0 A).
Definition nfa_to_dfa :=
  dfaI
    powerset_state
    powerset_s0
    [ pred X: powerset_state | existsb x: A, (x \in X) && nfa_fin A x]
    [ fun X a => \bigcup_(x | x \in X) finset (nfa_step A 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. We first prove that for every powerset state X and every state $x \in X$ we have that $\mathcal{L}_x(A) \subseteq \mathcal{L}_X(A_{det})$. Applying this to $X = \{s_0\}$ yields $\mathcal{L}(A) \subseteq \mathcal{L}(A_{det})$. We then show that for every powerset state X and word w with $w \in \mathcal{L}_X(A_{det})$ there exists a state x such that $x \in X$ and $w \in \mathcal{L}_x(A)$. For $X = \{s_0\}$, this shows $\mathcal{L}(A_{det}) \subseteq \mathcal{L}(A)$. Both proofs are done by induction on word. □

The formalization of this proof is straight-forward and follows exactly 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 automaton 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. It will later be useful to have automata that only contain reachable states. We define a procedure to extract the connected component from a given automaton.

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 definition, unreachable states have no influence on the language of an automaton because there is no run from the starting state that contains such a state. □

We make use of SSReflect's *connect* predicate to extract a sequence of all states reachable from s_0. 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 construct a new transition function that ensures transitions always end in reachable states. Theorem 4.2.1 is trivially solved by induction on word.

Add implementation

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. *We can decide emptiness of $\mathcal{L}(A)$ by computing the cardinality of A_c's set of final states, i.e.*

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

Proof. This is correct since $F_c = \emptyset \Leftrightarrow \mathcal{L}(A_c) = \emptyset$ and $\mathcal{L}(A_c) = \mathcal{L}(A)$ by theorem 4.2.1. □

Add implementation

4.4 Deciding Equivalence of Finite Automata

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

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

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

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

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

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

The decidability of this procedure follows directly from theorem 4.3.1. □

Add implementation

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 expressions. The structure of this proof is given by the inductive nature of regular expressions. For every constructor, we provide an equivalent automaton.

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.*

$$\mathcal{L}(A) = \bigcup_{f \in F} L^{|Q|}_{s_0,f} = \mathcal{L}(\sum_{f \in F} R^{|Q|}_{s_0,f}).$$

Proof. By definition, every $w \in \mathcal{L}(A)$ has a run that ends in some $f \in F$. Then, by definition, $w \in L^{|Q|}_{s_0,f}$.

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 ($i == j \wedge w = \varepsilon$, $i == j \wedge |w| = 1$, $i <> j \wedge |w| = 1$) 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

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

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 $L_{k,j}^{k+1}$. 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 we consider is given by the Myhill-Nerode theorem.

5.1 Definition

The following definitions (taken from [6]) will lead us to the statement of the Myhill-Nerode theorem. We assume $\equiv$ to be an equivalence relation on Σ^*, and L to be a language over Σ.

(i) $\equiv$ is **right congruent** if and only if for all $x, y \in \Sigma^*$ and $a \in \Sigma$,

$$x \equiv y \Rightarrow x \cdot a \equiv y \cdot a.$$

(ii) $\equiv$ **refines** L if and only if forall $x, y \in \Sigma^*$,

$$x \equiv y \Rightarrow (x \in L \Leftrightarrow y \in L).$$

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

$$\{[x] | x \in \Sigma^*\} \text{ is finite}$$

Definition 5.1.1. *A relation is Myhill-Nerode if and only it satisfies properties (i), (ii) and (iii).*

Fix everything below this line

Given a language L, the Myhill-Nerode relation $\approx_L$ is defined such that

$$\forall u, v \in \Sigma^*. \, u \approx_L v \iff \forall w \in \Sigma^*. \, u \cdot w \in L \Leftrightarrow v \cdot w \in L.$$

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 $\approx_L$ is of finite index.*

5.2 Finite Partitionings and Equivalence Classes

Coq does not have quotient types. We pair up functions and proofs for certain properties of those functions 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.,

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

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

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

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

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 $\approx_L$ must be of finite index since f's values directly correspond to equivalence classes. The image of f is finite. Therefore, $\approx_L$ is of finite index. □

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

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

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

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

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

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

[5] "S. C. Kleene". "representation of events in nerve nets and finite automata". *"Automata Studies"*, "1965".

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

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

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