# A Constructive Theory of Regular Languages in Coq

Christian Doczkal Jan-Oliver Kaiser Gert Smolka

*Published in Proc. of CPP 2013, Melbourne, Australia, LNCS 8307, Springer, 2013*

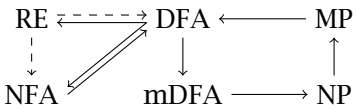
We present a formal constructive theory of regular languages consisting of about 1400 lines of Coq/Ssreflect. As representations we consider regular expressions, deterministic and nondeterministic automata, and Myhill and Nerode partitions. We construct computable functions translating between these representations and show that equivalence of representations is decidable. We also establish the usual closure properties, give a minimization algorithm for DFAs, and prove that minimal DFAs are unique up to state renaming. Our development profits much from Ssreflect's support for finite types and graphs.

## 1 Introduction

The theory of regular languages is a standard topic in the computer science curriculum [13, 10]. We are interested in an elegant and instructive formalization of this theory in constructive type theory. We prove Kleene's theorem [12], the Pumping Lemma, uniqueness of minimal deterministic automata, the Myhill-Nerode theorem [16, 17], and various closure properties of regular languages. For our formalization [7], we use the Ssreflect [9] extension to Coq [20] which features an extensive library with support for reasoning about finite structures such as finite types and finite graphs. Building on top of the Ssreflect infrastructure, we can establish all of our results in about 1400 lines of Coq, half of which are specifications. So our development may be considered a longish proof pearl.

The largest part of our formalization deals with translations between different representations of regular languages: regular expressions [12] (RE), deterministic finite automata [15] (DFA), minimal DFAs [11] (mDFA), nondeterministic finite automata [18] (NFA), Myhill partitions [16] (MP), and Nerode partitions [17]

(NP). We formalize all these representations and construct computable conversion functions between them. The conversion functions can be summarized by the following diagram:




The triangle corresponds to Kleene's theorem. For our representation of finite automata, we make use of the fact that in Coq, unlike HOL-based systems, types are first-class values. This allows us to formalize the sets of states of finite automata as finite types (i.e, types with finitely many inhabitants). Finite types are closed under many type constructors, and Ssreflect offers excellent support for finite types. So this representation is close to the usual mathematical definition and easy to work with.

For the conversion from regular expressions to finite automata (dashed lines), we need one construction for every constructor of regular expressions. By first establishing conversions between DFAs and NFAs, we can carry out each of these constructions on the automata model that fits best. The translation from DFAs to REs is based on Kleene's algorithm [12].

The rectangle in the diagram corresponds to the Myhill-Nerode theorem. Our constructive version of the Myhill-Nerode theorem differs from the usual formulation. We define Myhill and Nerode representations using functions from words into a finite type. We then give conversions to and from finite automata. This is similar to the formalization of the Myhill-Nerode theorem in Nuprl by Constable et al. [5], where decidability of Myhill-Nerode relations and finiteness of the corresponding quotient types are assumed.

Finite Myhill and Nerode partitions can be seen as abstract representations of DFAs where Nerode partitions correspond to minimal DFAs [13]. So the conversions from Nerode partitions to Myhill partitions and from Myhill partitions to DFAs are fairly direct. For the conversion from DFAs to Nerode partitions, we rely on automata minimization. For this, we prove correct a variant of Huffman's table filling algorithm for DFA minimization [11].

We also prove that regular languages are closed under images and preimages of homomorphisms, and we prove decidability of language equivalence for all representations. We prove our decidability results by defining, in Coq, functions to bool that decide the problem. This is sufficient since we work in the constructive logic of Coq without axioms, so every function into bool is total and computable.

## 1.1 Related Work

There are several recent publications concerned with the formalization of the theory of regular languages. Wu et al. [21] describe a formalization of the Myhill-Nerdode theorem in Isabelle/HOL based directly on regular expressions. The authors explain this unusual choice with limitations of Isabelle/HOL, which lacks a good graph library and does not allow quantification over types. This makes it difficult to define a type of automata that is easy to work with. Neither of these restrictions apply to Coq with Ssreflect. However, we could not find a formalization of automata theory in the literature that makes good use of the fact that types are fist-class values in Coq. For example, Braibant and Pous [4] formalize automata as part of a larger development building a certified decision procedure for Kleene algebras. They use numbered states, an approach rejected as "clunky" in [21].

There are a number of other papers describing certified implementations of decision procedures. These include decision procedures for regular expression equivalence [6, 1] in Coq or Matita. Berghofer et al. [3] formalize automata over bitstrings in Isabelle/HOL. Building on this formalization, they obtain a certified decision procedure for Presburger arithmetic. In these developments the focus is on efficiency. The formalizations are significantly longer than our development and formalize only those results relevant to the respective decision procedures. Kraus and Nipkow [14] develop a certified decision procedure for regular expression equivalence in Isabelle/HOL. Their formalization is very short, but they only show partial correctness. In contrast to most of the papers mentioned above, we are interested in simple proofs rather than practical executability. This gives us more freedom in our choice of representations.

## 1.2 Outline

After we introduce some notations and basic definitions, we explain our formalization of languages and give a decidable semantics for regular expressions. In Section 4, we prove Kleene's theorem. We also prove that equivalence of DFAs is decidable. In Section 5, we certify a minimization algorithm for DFAs and show that minimal DFAs are unique up to isomorphism. In Section 6, we prove our constructive Myhill-Nerode theorem. In Section 7, we prove that regular languages are closed under images and preimages of homomorphisms. In Section 8, we demonstrate how our Myhill-Nerode theorem can be used to show that a language is not regular.

## 2 Preliminaries

We review the basic mathematical definitions of formal languages and regular expressions. An **alphabet** $\Sigma$ is a finite set of symbols. The letters $a$, $b$, etc. denote symbols. For simplicity, we fix some alphabet $\Sigma$ throughout the paper. A **word** $w$ is a finite sequence of symbols from $\Sigma$. We write $w_n$ for the $n$-th symbol of $w$. The variables $u$, $v$ and $w$ always denote words and $\varepsilon$ denotes the empty word. A **language** is a set of words. We write $\Sigma^*$ for the language of all words and $\overline{L}$ for $\Sigma^* \setminus L$. We write $|w|$ to denote the **length** of the word $w$ and $v \cdot w$ or just $vw$ for the concatenation of $v$ and $w$. Here, $uv$ binds tighter than function application which in turn binds tighter than $\cdot$.

We consider simple **regular expressions** as defined by the following grammar

$$r, e := \emptyset \mid \varepsilon \mid a \mid r \cdot e \mid r + e \mid r^*$$

The language of a regular expression is defined as follows:

$$\mathcal{L}(\emptyset) = \emptyset \qquad \mathcal{L}(\varepsilon) = \{\varepsilon\} \qquad \mathcal{L}(a) = \{a\}$$

$$\mathcal{L}(r + e) = \mathcal{L}(r) \cup \mathcal{L}(e)$$

$$\mathcal{L}(r \cdot e) = \mathcal{L}(r) \cdot \mathcal{L}(e) = \{v \cdot w \mid v \in \mathcal{L}(r) \text{ and } w \in \mathcal{L}(e)\}$$

$$\mathcal{L}(r^*) = \mathcal{L}(r)^* = \{w_1 \cdot \ldots \cdot w_n \mid n \geq 0 \text{ and } \forall\, 0 < i \leq n.\ w_i \in \mathcal{L}(r)\}$$

## 3 Decidable Languages and Regular Expressions

Set theoretically, a language is just a set of words. Let char be a type of characters. We write word char for sequences over char and represent languages as predicates of type word char → Prop. Note that languages are not necessarily decidable, i.e., there are languages $L$ for which we can prove neither $w \in L$ nor $w \notin L$. Regular languages, however, are always decidable. So for most of our development we use decidable languages word char → bool, which are more convenient to work with. We formalize decidable languages using Ssreflect's boolean predicates.

**Definition** dlang char := pred (word char).

Note that our representation of languages as predicates is intensional, i.e., equivalent languages are not necessarily equal. We state equivalences between decidable languages using Ssreflect's extensional equality operator, which satisfies

L1 =i L2 ↔ ∀ w, (w ∈ L1) = (w ∈ L2)

Here, ∈ is Ssreflect's generic membership operator for everything that can be seen as a boolean predicate. All our constructions respect this equivalence, so for our informal explanations we consider equivalent languages as equal.

We assign decidable languages to every representation of regular languages. To do this for regular expressions, we have to show that decidable languages are closed under all regular operations. Proving this in Coq amounts to defining $\emptyset, \varepsilon, a, \cdot, +$, and $^*$ as operators on decidable languages. We use the definitions from Coquand and Siles [6] who also work with decidable languages. All operators but $\cdot$ and $^*$ are easily defined. The operator for $\cdot$ is

```
Definition conc (L1 L2: dlang char) : dlang char :=
  fun v ⇒ [exists i : 'I_(size v).+1, L1 (take i v) && L2 (drop i v)].
```

where 'I_(size v).+1 is the type of natural numbers smaller than (size v) + 1. This type has only finitely many inhabitants and is therefore called a finite type. Decidability is preserved by quantification over finite types and Ssreflect's boolean quantifier [exists x : T, p x] yields a boolean result provided that T is a finite type and p is a boolean predicate. Hence, conc L1 L2 is a decidable language.

For the concatenation operator, we prove the correctness lemma

```
Lemma concP {L1 L2 : dlang char} {w : word char} :
  reflect (∃ w1 w2, w = w1 ++ w2 ∧ w1 ∈ L1 ∧ w2 ∈ L2) (w ∈ conc L1 L2).
```

which reflects[1] the boolean membership statement into a Coq proposition. The definitions of the remaining operators and the associated correctness lemmas can be found in the file `regexp.v` of our formalization [7].

We represent regular expressions using an inductive type. Having defined all the regular operations on languages, we can associate a decidable language to every regular expression.

```
Fixpoint re_lang (e : regexp char) : dlang char :=
  match e with
  | Void ⇒ void char
  | Eps ⇒ eps char
  | Atom x ⇒ atom x
  | Star e1 ⇒ star (re_lang e1)
  | Plus e1 e2 ⇒ plus (re_lang e1) (re_lang e2)
  | Conc e1 e2 ⇒ conc (re_lang e1) (re_lang e2)
  end.
```

**Theorem 3.1** The matching problem for regular expressions is decidable.

**Proof** This is an immediate consequence of defining the semantics of regular expressions in terms of decidable languages.

We now call a general language regular if it is equivalent to the language of some regular expression.

[1] For P : Prop and p : bool, the statement reflect P p asserts that the Coq proposition P and p = true are logically equivalent.

```
Definition regular char (L : word char → Prop) := ∃ e : regexp char, ∀ w, L w ↔ w ∈ e.
```

Defining regularity on general languages has the advantage that one can prove regularity by giving a regular expression or a finite automaton without first proving that the language is decidable.

## 4 Kleene's Theorem

While regular expressions can be seen as the natural characterization of regular languages, finite automata can be seen as an operational characterization of the same class of languages. Several theorems about regular languages can be proven more easily using automata rather than regular expressions. This includes the closure of regular languages under complement and the Myhill-Nerode theorem. We formalize nondeterministic and total deterministic automata and show that both have the same expressive power as regular expressions.

**Definition 4.1** · A **nondeterministic finite automaton** (NFA) is a tuple $(Q, s, F, \delta)$ where $Q$ is a finite set of states, $s \in Q$ is the starting state, $F \subseteq Q$ is the set of accepting states, and $\delta \subseteq (Q \times \Sigma) \times Q$ is the transition relation.

· A **deterministic finite automaton** (DFA) is a tuple $(Q, s, F, \delta)$ as above, except that $\delta : Q \times \Sigma \to Q$ is a total function. □

We formalize NFAs and DFAs as two separate record types.

```
Record nfa : Type := {
  nfa_state :> finType;
  nfa_s : nfa_state;
  nfa_fin : pred nfa_state;
  nfa_trans : nfa_state → char → nfa_state → bool }.
```

The definition of dfa is the same, except that the transition function dfa_trans is a function of type dfa_state → char → dfa_state.

For these definitions, we make use of the fact that types are first-class objects in Coq. This allows us to represent the set of states as a type. We require that the type of states has a finType structure [8], i.e., is a finite type. The collection of types with a finType structure can be thought of as a type class. It is closed under the product, sum, option, and set type constructors. That means that if we have finType structures for T and T', the type checker can infer finType structures for $T * T'$, $T + T'$, option T, and {set T}, the type of sets over T. So the finType structures of all our automata constructions are inferred automatically. The annotation :> for nfa_state registers nfa_state as a coercion. So if A:nfa, x:A means that x is a state of A.

We follow [13] and define acceptance for all states of an automaton by recursion on words.

```
Fixpoint nfa_accept (A : nfa) (x : A) w :=
  if w is a :: w' then [exists y, nfa_trans A x a y && nfa_accept A y w']
                  else x ∈ nfa_fin A.

Fixpoint dfa_accept (A : dfa) (x : A) w :=
  if w is a :: w' then dfa_accept A (dfa_trans A x a) w' else x ∈ dfa_fin A.
```

The language of an automaton is the set of words accepted by the starting state. Every DFA can easily be converted into an NFA accepting the same language. For the converse direction we formalize the usual powerset construction. Thus we obtain:

**Theorem 4.2** For every NFA (DFA) we can construct a DFA (NFA) accepting the same language.

## 4.1 Regular Expressions to Finite Automata

Our next result about finite automata is the construction of an automaton for every regular expression. For this we need constructions on finite automata corresponding to constructors of regular languages. Since `dfa_accept` does not use existential quantification, DFAs are generally easier to work with. However, some constructions become much easier when done on NFAs.

Since we have already established conversions between NFAs and DFAs, we can carry out each construction on the automata model that fits best. We define the constructions for $A^*$, $A \cdot B$, and $'a'$ on NFAs and we define the constructions for $\emptyset$, $\varepsilon$, and $A + B$ on DFAs. All six constructions are fairly straightforward. Our NFA constructions differ slightly from Kozen's [13] since our NFAs do not admit $\varepsilon$-transitions. Whenever one would usually use an $\varepsilon$-transition from a state $x$ to a state $y$, we instead duplicate all incoming transitions from $x$ as incoming transitions of $y$. See the file `automata.v` [7] for details.

**Theorem 4.3** For every regular expression $r$, we can construct a DFA accepting the same language.

**Proof** Induction on $r$ using the respective constructions on automata.

For DFAs it is also easy to show closure under complement and intersection. In fact, the DFA for $A \cap B$ and the DFA for $A + B$ used above are both instances of one generic construction for binary boolean operations. Further, we can give a function that decides whether the language of a DFA is empty.

```
Definition dfa_lang_empty A := [forall (x | reachable A x), x ∉ dfa_fin A].
```

The function simply checks that none of the states of `A` which are reachable from

the starting state are final states of A. Here, reachable is defined with respect to the reflexive transitive closure of the relation below:

```
Definition dfa_trans_some (x y : A) := [exists a, dfa_trans x a == y].
```

The reflexive transitive closure of a decidable relation over a finite type is again a decidable relation and this construction is contained in the Ssreflect libraries. So we obtain:

**Theorem 4.4** 1. Language emptiness for DFAs is decidable.

2. Language equivalence for DFAs is decidable.

**Proof** (1) is decided by dfa_lang_empty. Part (2) reduces to (1).

Note that decidability of language equivalence for DFAs implies decidability of language equivalence for all representations that can be translated to DFAs. We will show that this is the case for all representations we consider.

## 4.2 Finite Automata to Regular Expressions

We now show that we can construct a regular expression for every deterministic finite automaton. We use Kleene's algorithm [12], because we think it is easiest to formalize. For the rest of this section, we assume that we are given a DFA $A = (Q, s, F, \delta)$.

**Definition 4.5** Let $w$ be a word and $x \in Q$. We call the state sequence $x_1 \dots x_n$ the **run** from $x$ on $w$, written $\mathrm{run}(x, w)$, if $x \overset{w_1}{\to} x_1 \dots \overset{w_{|w|}}{\to} x_{|w|}$ where $y \overset{a}{\to} z$ abbreviates $\delta(y, a) = z$. We define $\hat{\delta}(x, w)$ to be the last element of $x :: \mathrm{run}(x, w)$ and write $\hat{\delta}_s$ for $\lambda w.\ \hat{\delta}(s, w)$. □

Based on the notion of run, we can define the languages of runs from one state to another. Restricting the set of states that may be traversed in between, we obtain the following indexed collection of languages:

**Definition 4.6** Let $X \subseteq Q$, $x, y \in Q$, and $w$ a word. $w \in L^X_{x,y}$ iff (1) $\hat{\delta}(x, w) = y$ and (2) all states of $\mathrm{run}(x, w)$ except possibly the last are contained in $X$. □

**Lemma 4.7** $\mathcal{L}(A) = \bigcup_{x \in F} L^Q_{s,x}$

According to Lemma 4.7, it suffices to construct regular expressions for $L^Q_{s,x}$ for the various states $x \in F$ in order to obtain a regular expression for $\mathcal{L}(A)$. We recursively solve the problem for all languages $L^Q_{x,y}$ by successively removing states from $Q$. Once we reach the empty set of states, we can directly give a regular expression.

**Definition 4.8** Let $x, y \in Q$, then

$$R^{\emptyset}_{x,y} \stackrel{def}{\equiv} (\text{if } x = y \text{ then } \varepsilon \text{ else } \emptyset) + \sum_{\substack{a \in \Sigma \\ \delta(x,a)=y}} a$$

□

**Lemma 4.9** $\mathcal{L}\left(R^{\emptyset}_{x,y}\right) = L^{\emptyset}_{x,y}$

Now consider the case of a nonempty set $X \subseteq Q$, where $z \in X$ and $w \in L^X_{x,y}$. Then $\text{run}(x, w)$ may reach $z$, come back to $z$ an arbitrary number of times, and then end in $y$. Alternatively, the run may not reach $z$ at all. This motivates the following lemma:

**Lemma 4.10** Let $x, y, z \in Q$ and $X \subseteq Q$, then

$$w \in L^{\{z\}\cup X}_{x,y} \iff w \in L^X_{x,z} \cdot \left(L^X_{z,z}\right)^* \cdot L^X_{z,y} + L^X_{x,y}$$

The formalization of Lemma 4.10 is one of the more involved parts of our development. Writing delta for $\hat{\delta}$, we formalize $L^X_{x,y}$ as follows:

```
Definition L (X : {set A}) (x y : A) :=
    [pred w | (delta x w == y) && abl (mem X) (dfa_run x w)].
```

Here, abl (mem X) (dfa_run x w) checks the second condition of Definition 4.6. Showing the direction from right to left is relatively easy. We prove the converse direction by induction on $|w|$. The essential lemma for this direction is:

```
Lemma L_split X x y z w : w ∈ L^(z |: X) x y →
  w ∈ L^X x y ∨ ∃ w1 w2, w = w1 ++ w2 ∧ size w2 < size w
                       ∧ w1 ∈ L^X x z ∧ w2 ∈ L^(z |: X) z y.
```

which is itself proved by induction on $w$. The Notation z |: X stands for $\{z\} \cup X$. Following Lemma 4.9 and Lemma 4.10, we can give a recursive procedure R such that the language of R^X x y is L^X x y.

```
Function R (X : {set A}) (x y : A) {measure (fun X ⇒ #|X|) X} :=
  match [pick z ∈ X] with
  | None ⇒ R0 x y
  | Some z ⇒ let X' := X :\ z in
      Plus (Conc (R X' x z) (Conc (Star (R X' z z)) (R X' z y))) (R X' x y)
  end.
```

The definition employs Coq's Function command, which allows the definition of functions by size recursion. In our case, the measure is the size of the set X. The expression [pick z ∈ X] evaluates to None if X is empty and to Some z with z ∈ X otherwise. Due to the match on the pick expression, Coq is, at the time of writing, not capable of generating the functional induction principle for R.

However, this does not pose a problem, since the correspondence between R and L can be proved directly by induction on the size of X.

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

Thus we have:

**Theorem 4.11** For every automaton A, we can construct a regular expression accepting $\mathcal{L}(A)$.

**Corollary 4.12** Let $r$ and $e$ be regular expressions. We can construct regular expressions accepting $\mathcal{L}(r) \cap \mathcal{L}(e)$ and $\overline{\mathcal{L}(r)}$.

# 5 Minimization

We now construct a minimization function for DFAs. We follow Kozen's presentation [13] of Huffman's table filling algorithm [11]. We fix some DFA $A = (Q, s, F, \delta)$.

**Definition 5.1** Let $x, y \in Q$. The **collapsing relation on** $A$ is defined as follows:

$$x \approx y \stackrel{def}{\equiv} \forall w.\, \delta(x, w) \in F \iff \delta(y, w) \in F$$ □

Minimization merges every equivalence class of the collapsing relation into a single state. To construct this quotient automaton in Coq, we need to show that the collapsing relation is decidable. Once we have a boolean reflection collb : A → A → bool of the collapsing relation, the quotient construction follows a generic pattern.

The construction makes use of the fact that there is a constructive choice operator for finite types. Consider, for instance, a decidable equivalence relation e over a type T with choice operator. We use this choice operator to get a canonical element[2] of every equivalence class of e.

**Definition** repr x := choose (e x) x.

We call repr x the representative of the equivalence class of x with respect to e. The function repr is idempotent. Thus, the quotient of T modulo e can be defined as follows:

**Definition** quot := { x : T | x == repr x }.

The type quot is a sigma type, i.e., the type of dependent pairs of elements x and proofs of x == repr x. In particular, this type is finite if T is finite, since x == repr x has at most one proof and, thus, quot has at most as many elements as T. Using

[2] If p y = true, the result of choose p y satisfies p but does not depend on y.

repr we can also define a function class : T → quot corresponding to the function $\lambda x.[x]_e$. The first projection of quot, written val, allows us to obtain the canonical representative of the class $[x]_e$. Taking e to be collb we can define the quotient automaton as follows:

```
Definition minimize : dfa := {|
  dfa_s := class (dfa_s A);
  dfa_trans x a := class (dfa_trans (val x) a);
  dfa_fin := [pred x | val x ∈ dfa_fin A ] |}.
```

To compute collb, we compute its complement, the distinguishable states, using a fixpoint construction. Final and non-final states are distinguishable using the empty word.

```
Definition dist0 : {set A*A} := [set x | (x.1 ∈ dfa_fin A) != (x.2 ∈ dfa_fin A)].
```

We also mark those pairs as distinguishable that transition to an already distinguishable pair of states.

```
Definition distS (dist : {set A*A}) :=
  [set x | [exists a, (dfa_trans x.1 a, dfa_trans x.2 a) ∈ dist]].
```

Now we can define a monotone function one_step_dist : {set M * M} → {set M * M} corresponding to one pass through Huffman's algorithm [11].

```
Definition one_step_dist dist := dist0 ∪ distS dist.
```

Its least fixpoint, computed by iterating the function sufficiently often on the empty set, is exactly the set of distinguishable pairs. This finishes the minimization construction. At this point, we can show:

```
Lemma minimize_correct A : dfa_lang (minimize A) =i dfa_lang A.
```

```
Lemma minimize_size A : #|minimize A| ≤ #|A|.
```

For connected DFAs the result of minimization is indeed minimal and minimal DFAs are unique up to isomorphism, i.e, up to a renaming of the states. In this context, a connected DFA is a DFA $A = (Q, s, F, \delta)$ where $\hat{\delta}_s$ is surjective. Surjectivity of $\hat{\delta}_s$ allows us to define a partial inverse $\hat{\delta}_s^{-1} : Q \to \Sigma^*$ such that $\hat{\delta}_s(\hat{\delta}_s^{-1}x) = x$. For this we exploit that there is also a choice operator for the countable type of words. In fact our inverse construction for $\hat{\delta}_s$ is just an instance of a generic inverse construction for surjective functions from types with choice operator to types with decidable equality.

```
Definition cr {X : choiceType} {Y : eqType} {f : X → Y} (Sf : surjective f) y : X :=
  xchoose (Sf y).
```

We call cr Sf x (with types as above) the **canonical representative** of x. The construction uses xchoose, a stronger variant of constructive choice, which for a decidable predicate p turns a proof of ∃ x, p x into an element satisfying p.

We now show that all connected and collapsed (i.e., the collapsing relation is the identity) DFAs are isomorphic. Consider two collapsed DFAs A and B accepting the same language and a proof A_conn : connected A for A and likewise for B. We use cr to define an isomorphism between A and B.

```
Definition iso (x : A) : B := delta (dfa_s B) (cr A_conn x).
```

To show that iso is a bijection we define its inverse iso_inv by swapping the role of A and B in the definition of iso and show that iso and iso_inv cancel each other in both directions. The proofs make use of the following fact about the interaction of delta and iso:

```
Lemma delta_iso w x : delta (iso x) w ∈ dfa_fin B = (delta x w ∈ dfa_fin A).
```

Using the fact that both automata are fully collapsed, we can show that iso is not just a bijection but also respects the structure of the automata. Thus, we have shown:

**Theorem 5.2** Let $A = (Q_1, s_1, F_1, \delta_1), B = (Q_2, s_2, F_2, \delta_2)$ be collapsed and connected DFAs. If $\mathcal{L}(A) = \mathcal{L}(B)$, then there exists a bijection $i : Q_1 \to Q_2$ satisfying

$$\begin{aligned} \forall q \in Q_1.\ i(\delta_1(q,a)) &= \delta_2(i(q),a) \\ \forall q \in Q_1.\ i(q) \in F_2 &\iff q \in F_1 \\ i(s_1) &= s_2 \end{aligned}$$

It is easy to show that minimization preserves connectedness and yields collapsed DFAs. In particular, this entails that the result of minimizing a connected automaton is indeed minimal and that minimal DFAs are unique up to isomorphism.

## 6 Myhill-Nerode Theorem

Myhill [16] and Nerode [17] relations characterize regular languages in terms of simple algebraic properties. We now define two additional representations of regular languages: Myhill partitions and Nerode partitions. Our constructive version of the Myhill-Nerode theorem then consists of three conversion functions: from Nerode partitions to Myhill partitions, from Myhill partitions to DFAs, and from minimal DFAs to Nerode partitions.

**Definition 6.1** Let $\equiv \subseteq \Sigma^* \times \Sigma^*$ be an equivalence relation. The **partition** $(\mathcal{I}, E)$ with $\mathcal{I}$ a finite set and $E : \Sigma^* \to \mathcal{I}$ surjective **represents** $\equiv$ if

$$\forall u\, v.\ E\, u = E\, w \iff u \equiv w$$

We call $\mathcal{I}$ the **index set** and $E$ the **representation function**. Further, $E(w)$ represents $[w]_{\equiv}$, the equivalence class of $w$ with respect to $\equiv$. □

Every partition $(\mathcal{I}, E)$ represents some equivalence relation of finite index. We phrase the Myhill and Nerode conditions directly in terms of partitions.

**Definition 6.2** Let $P = (\mathcal{I}, M)$ be a partition. $P$ is a **Myhill partition** for $L$, if $M$

1. is **right congruent:** $\forall u \in \Sigma^*\ v \in \Sigma^*\ a \in \Sigma.\ M\,u = M\,v \Rightarrow M\,ua = M\,va$
2. **refines** $L$: $\forall u \in \Sigma^*\ v \in \Sigma^*.\ M\,u = M\,v \Rightarrow (u \in L \iff v \in L)$ □

**Definition 6.3** Let $P = (\mathcal{I}, N)$ be a partition. $P$ is a **Nerode partition** for $L$ if

$$\forall u \in \Sigma^*\ v \in \Sigma^*.\ N\,u = N\,v \iff \forall w \in \Sigma^*.\ (uw \in L \iff vw \in L)$$ □

We refer to the representation functions of Myhill partitions as Myhill functions and similarly for Nerode partitions. We formalize finite partitions using records:

```
Record finPar := {
  finpar_classes : finType;
  finpar_fun :> word → finpar_classes;
  finpar_surj : surjective finpar_fun }.
```

In addition to finpar_fun, we manually register finpar_classes as a second coercion. So if E:finPar, we write x:E to mean that $x$ is in the index set just as we did for the states of automata, but we can also write E w for the class of a word w.

We then formalize Myhill and Nerode partitions as more constrained versions of the above type. For example, Nerode partitions for $L$ are defined as follows:

```
Definition nerode (X : eqType) (L : dlang char) (E : word → X) :=
  ∀ u v, E u = E v ↔  ∀ w, (u++w ∈ L) = (v++w ∈ L).

Record nerodePar L := {
    nerode_par :> finPar;
    nerodeP : nerode L nerode_par }.
```

We now define the translation functions we need for our Myhill-Nerode theorem. Myhill partitions can be seen as abstract representations of connected DFAs and Nerode partitions can be seen as abstract representations of minimal DFAs.

The translation from Nerode partitions to Myhill partitions is particularly easy. The representation function of a Nerode partition for a language $L$ refines $L$ and is right congruent. Thus the representation function of a Nerode partition can also serve as representation function for a Myhill partition.

For the direction from Myhill partitions to DFAs consider some Myhill partition $(\mathcal{I}, M)$ representing a Myhill relation $\equiv$ for some language $L$. We construct an automaton $A$ with $Q := \mathcal{I}$ and $s := M\,\varepsilon$. For the final states and the transition function, we make use of the fact that $M : \Sigma^* \to \mathcal{I}$ is surjective. Due to

surjectivity, every $x \in \mathcal{I}$ represents a class $[w]_{\equiv}$ for some $w$. Using cr we can now define a transition function on $\mathcal{I}$ which in terms of the represented classes satisfies $\delta([w]_{\equiv}, a) = [wa]_{\equiv}$.

```
Definition fp_trans (E : finPar) (x : E) a := E (cr E x ++ [:: a]).
```

The conversion from Myhill partitions to DFAs is then defined as follows:

```
Definition myhill_to_dfa L (M : myhillPar L) :=
  {| dfa_s := M [::]; dfa_fin x := cr M x ∈ L; dfa_trans := @fp_trans M |}.

Lemma myhill_to_dfa_correct L (M : myhillPar L) : dfa_lang (myhill_to_dfa M) =i L.
```

For the conversion from DFAs to Nerode partitions we can rely on our minimization algorithm. Hence, it is sufficient to convert minimal DFAs to Nerode partitions. Consider, for instance, a minimal DFA $A = (Q, s, F, \delta)$. Since $A$ is minimal and thus connected, the function $\hat{\delta}_s : \Sigma^* \to Q$ is surjective. Further $A$ is collapsed and therefore $(Q, \hat{\delta}_s)$ is a Nerode partition. Thus we have:

**Theorem 6.4**

1. For every Nerode partition we can construct an equivalent Myhill partition.
2. For every Myhill partition we can construct an equivalent DFA.
3. For every DFA we can construct an equivalent Nerode partition.

# 7 Closure under Homomorphisms

In Section 4.2, we have shown that regular languages are closed under intersection and complement. We now show that regular languages are closed under preimages and images of homomorphisms. For this section we assume a second alphabet $\Gamma$.

**Definition 7.1** Let $h : \Sigma^* \to \Gamma^*$ be a function.

- $h$ is a **homomorphism** if $h\,uv = h\,u \cdot h\,v$ for all $u \in \Sigma^*$ and $v \in \Sigma^*$.
- The **preimage** of $L$ under $h$ is $h^{-1}(L) := \{w \mid h\,w \in L\}$
- The **image** of $L$ under $h$ is $h(L) := \{v \mid \exists w \in L.\ h\,w = v\}$ □

The closure under taking the preimage of a homomorphism is easy to show. We use Kozen's [13] construction on DFAs.

The closure of regular languages under taking the image of a homomorphism is more interesting. Unlike all the closure properties of regular languages we have shown so far, it is not a closure property of decidable languages in general. This means, that we cannot define an image operator on decidable languages. We can only express the image as a predicate in Prop.

```
Definition image (h : word char → word char') (L : word char → Prop) v :=
  ∃ w, L w ∧ h w = v.
```

For a homomorphism $h$, Kozen [13] gives a construction of a regular expression $e_h$ from a regular expression $e$ satisfying $\mathcal{L}(e_h) = h(\mathcal{L}(e))$. The construction works by replacing all atoms $a$ in e with the string $h\,a$. We can prove this construction correct, but we can only state the correctness as a reflection lemma and not as a quantified boolean equation as we did for all the other constructions so far.

```
Lemma re_imageP e v : reflect (image h (re_lang e) v) (v ∈ re_image e).
```

Once we abstract away the concrete constructions, this difference disappears and we obtain:

```
Lemma preim_regular (char char' : finType) (h : word char → word char') L :
  homomorphism h → regular L → regular (preimage h L).

Lemma im_regular (char char' : finType) (h : word char → word char') L :
  homomorphism h → regular L → regular (image h L).
```

**Theorem 7.2** Regular languages are closed under taking preimages and images of homomorphisms.

## 8 Proving Languages Non-Regular

So far, we have formalized a number of constructions that can be used to prove that a language is regular. One option to prove that a language is not regular is using the Pumping Lemma, which is included in our formalization. More interesting in our constructive setting is the use of Nerode partitions. If we want to prove that a language $L$ is not regular, we can assume that $L$ is regular. This provides us with a corresponding decidable language $L'$ and, using our translation functions, with a Nerode partition for $L'$.

```
Lemma regularE (L : word char → Prop) : regular L →
  ∃ L' : dlang char, (∀ w, L w ↔ w ∈ L') ∧ inhabited (nerodePar L').
```

Hence, it is sufficient to have a reasoning principle to show non-regularity of decidable languages. One such criterion is the existence of an infinite collection of words that are not related by the Nerode relation.

```
Lemma nerodeIN (f : nat → word char) (L : dlang char) :
  (∀ n1 n2, (∀ w, (f n1 ++ w ∈ L) = (f n2 ++ w ∈ L)) → n1 = n2) →
  ~ regular L.
```

We use this principle to prove that $\{w \mid \exists n.\ w = a^n b^n\}$ is not regular.

So for proofs of non-regularity, the restriction of Nerode partitions to decidable languages is irrelevant. For proofs of regularity, it does pose a restriction,

and this restriction is unavoidable in a constructive setting. Consider some independent proposition $P$, i.e., some $P$ for which we can prove neither $P$ nor $\neg P$. Then regularity of the language $L := \{w \mid P\}$ is equivalent to the unprovabe proposition $P \vee \neg P$. However, we can still prove that $L$ has exactly one Nerode class. So except for the decidability requirement on the language itself, we have a Nerode partition and thus a proof of regularity. This also shows that there are some languages on which the constructive interpretation of regularity differs from the set theoretic interpretation.

## 9 Conclusion

We have formalized a number of fundamental results about regular languages. Our selection of results corresponds roughly to the first part of Kozen's "Automata and Computability" [13]. We added the uniqueness result for minimal DFAs and skipped NFAs with $\varepsilon$-transitions, stopping right before the definition of 2DFAs.

Concerning $\varepsilon$-NFAs, we came to the conclusion that the added flexibility is not worth the effort. In fact, we had a formalization of $\varepsilon$-NFAs. However, in our formalization the correctness proof for the conversion function from $\varepsilon$-NFAs to NFAs was just reflexivity and thus not interesting. So the only advantage of working with $\varepsilon$-NFAs would be more compact definitions, but at the cost of having to consider, in proofs, possible sequences of $\varepsilon$-transitions at every transition step.

A more interesting addition to our development would be 2DFAs. Since 2DFAs may move back and forth on the input word and even have infinite runs, their language cannot be defined by a simple recursion on the input word. Instead, the boolean acceptance predicate would be computed using a finite fixpoint construction similar to the one we employed in the minimization algorithm.

To make our formalization more instructive, we obtain our results with many small lemmas. This way at least the overall structure of the proofs can be understood without stepping through the proofs. In total, we prove about 170 lemmas and almost 50% of the 1400 lines of our development are specifications.

All of our proofs are carried out in the constructive type theory of Coq, and there are a number of places where we have to deviate from the textbook presentation [13] of the material we formalize. We have to show at several places that the predicates and relations we define are decidable, which is supported very well by Ssreflect.

For Kleene's Theorem, staying constructive did not cause any difficulties, since the textbook proof [13] of this theorem is also constructive. For automata minimization, the lack of general quotient types in Coq forces us to compute the collapsing relation before we can define the quotient automaton. However,

the collapsing algorithm is interesing in its own right. For the Myhill-Nerode Theorem, we represent equivalence relations of finite index by making the finite set of equivalence classes explicit in the form of a finite type. This turns Myhill and Nerode partitions into objects we can compute with. Consequently, we have to rely on minimization to obtain Nerode partitions.

Many of the constructions we use in our development are applicable in a wide range of situations. Notably, this includes finite types, constructive quotients, and finite fixpoints. While we are certainly not the first to use these techniques, we believe they deserve a wider recognition. Since the material we formalize is fairly standard, we hope that our proofs can serve as examples for teaching theory development in Coq.

### Acknowledgments.

We thank the anonymous reviewers of the preliminary versions of this paper for their helpful comments.

## References

[1] Asperti, A.: A compact proof of decidability for regular expression equivalence. In: Beringer, L., Felty, A.P. (eds.) ITP. LNCS, vol. 7406, pp. 283–298. Springer (2012)

[2] Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.): Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, LNCS, vol. 5674. Springer (2009)

[3] Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: Berghofer et al. [2], pp. 147–163

[4] Braibant, T., Pous, D.: Deciding kleene algebras in coq. Logical Methods in Computer Science 8(1) (2012)

[5] Constable, R.L., Jackson, P.B., Naumov, P., Uribe, J.C.: Constructively formalizing automata theory. In: Plotkin, G.D., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction. pp. 213–238. The MIT Press (2000)

[6] Coquand, T., Siles, V.: A decision procedure for regular expression equivalence in type theory. In: Jouannaud, J.P., Shao, Z. (eds.) CPP. LNCS, vol. 7086, pp. 119–134. Springer (2011)

[7] Doczkal, C., Kaiser, J.O., Smolka, G.: Formalization accompanying this paper, `http://www.ps.uni-saarland.de/extras/cpp13/`

[8] Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer et al. [2], pp. 327–342

[9] Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA (2008), http://hal.inria.fr/inria-00258384

[10] Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation - international edition (2. ed). Addison-Wesley (2001)

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

[12] Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Shannon and McCarthy [19], pp. 3–42

[13] Kozen, D.: Automata and computability. Undergraduate texts in computer science, Springer (1997)

[14] Krauss, A., Nipkow, T.: Proof pearl: Regular expression equivalence and relation algebra. J. Autom. Reasoning 49(1), 95–106 (2012)

[15] Moore, E.F.: Gedanken-experiments on sequential machines. In: Shannon and McCarthy [19], pp. 129–153

[16] Myhill, J.R.: Finite Automata and the Representation of Events. Tech. Rep. WADC TR-57-624, Wright-Paterson Air Force Base (1957)

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

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

[19] Shannon, C., McCarthy, J. (eds.): Automata Studies. Princeton University Press (1956)

[20] The Coq Development Team: http://coq.inria.fr

[21] Wu, C., Zhang, X., Urban, C.: A formalisation of the Myhill-Nerode theorem based on regular expressions (proof pearl). In: van Eekelen, M.C.J.D., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP. LNCS, vol. 6898, pp. 341–356. Springer (2011)