Lean Formalization of DFA Minimization

1 Project overview

This repository contains a Lean 4 formalization of **deterministic finite automata (DFAs)** and the classical **minimization theorem**: given a DFA, construct a **canonical minimal DFA** that accepts the same language, and prove it is **unique up to isomorphism** (here: bijective DFA morphism).

Mathlib already defines DFAs and basic operations on them, but it does not include standard automata-theory infrastructure such as **morphisms**, **accessibility**, and **minimization**. This project fills that gap, and also develops **computable** versions of the key definitions so that the constructions can be executed when the input DFA is effectively finite.

2 Lean 4 and Mathlib (minimal background)

Lean 4 is an interactive theorem prover and functional programming language. In Lean, definitions are written as programs, and theorems are written as types whose inhabitants are proofs. Mathlib is the main community library of formalized mathematics and related theory, including some computability and automata definitions.

This document includes Lean snippets. You do not need to read them as executable code. The main point is that the definitions and proofs are fully machine checked.

3 DFAs in Mathlib

A DFA is parameterized by an alphabet type α and a state type σ:

- α is the type of input symbols
- σ is the type of states

In Mathlib, σ is not required to be finite. This is convenient for theory, but finiteness (and computability) must be added later when we want executable algorithms.

A DFA has three fields:

- `step :` is the transition function
- `start :` is the start state
- `accept : Set` is the set of accepting states

```
structure DFA ( : Type u) ( : Type v) where
  step :
  start :
  accept : Set
```

3.1 Evaluating a DFA on a word

A word is represented as a list `List` . The evaluation function folds transitions along the list:

```
def evalFrom (M : DFA  ) (s : ) : List    := List.foldl M.step s
def eval (M : DFA  ) : List    := M.evalFrom M.start
```

A DFA accepts a word if the final state lies in the accepting set:

```
def acceptsFrom (M : DFA  ) (s : ) (w : List ) : Prop := M.evalFrom s w  M.accept
def accepts (M : DFA  ) : Language  := { w | M.acceptsFrom M.start w }
```

Here `Language` is a set of words over α. So `M.accepts` is the language recognized by `M`.

4 DFA morphisms

A central ingredient in minimization is a notion of structure-preserving map between DFAs.

A morphism from `M` to `N` (same alphabet) is a function on states that preserves:

(1) the start state

(2) acceptance

(3) behavior on all words (equivalently, it commutes with evaluation)

In Lean we package this as a structure whose fields include the function plus proofs of the preservation properties:

```
structure Hom (M : DFA  ) (N : DFA  ) where
  toFun :
  map_start : toFun M.start = N.start
  map_accept (q : ) : q  M.accept  toFun q  N.accept
  map_step (q : ) (w : List ) :
    toFun (M.evalFrom q w) = N.evalFrom (toFun q) w

infixr:25 "  " => Hom
```

Intuition: `map_step` says "if you run `M` from state `q` on word `w` and then map the resulting state, you get the same result as mapping `q` first and then running `N` on `w`."

4.1 Morphisms preserve recognized language

A standard theorem is that if there is a morphism `f :  M  N`, then the DFAs recognize the same language.

The proof is essentially one line of reasoning written out carefully: a word is accepted by `M` iff the reached state is accepting; `f` preserves (i) which state you reach and (ii) which states are accepting.

```
theorem Hom.pres_lang {M : DFA  } {N : DFA  } (f : M  N) :
    M.accepts = N.accepts := by
  ext w
  simp only [accepts, acceptsFrom, Set.mem_setOf_eq]
  constructor
  · intro h
    rw [f.map_accept] at h
    rw [f.map_step M.start w] at h
    rw [f.map_start] at h
    exact h
  · intro h
    rw [f.map_accept, f.map_step M.start w, f.map_start]
    exact h
```

We also define:

- **surjective morphisms**: the underlying function is surjective (notation in the codebase: `N  M`)

- **equivalences**: bijective morphisms (notation in the codebase: `M  N`)

Surjective morphisms define a preorder that matches the standard "can be obtained by quotienting states" notion.

5 What is DFA minimization?

Different DFAs can accept the same language. Minimization produces a DFA with no redundant structure, and it is canonical up to renaming states.

A common informal definition is "fewest states," but in automata theory it is convenient to define minimality using morphisms:

> For DFAs over the same alphabet, define $M \leq N$ iff there exists a **surjective** morphism $N \twoheadrightarrow M$.

This captures the idea that M is obtained from N by collapsing states in a way that preserves behavior.

A **minimal DFA** for a language L is a DFA M recognizing L such that $M \leq N$ for every DFA N recognizing L. Such an M is unique up to equivalence.

This project formalizes this preorder, constructs a minimal DFA, proves it recognizes the same language, and proves uniqueness up to equivalence.

6 The construction used in this project

There are multiple minimization algorithms (for example, Hopcrofts partition refinement algorithm). This project uses the classical "quotient by Nerode equivalence" construction because it matches the theoretical statement of the Myhill–Nerode theorem and is proof-friendly in Lean.

6.1 Nerode equivalence on states

Fix a DFA `M : DFA  `. For a state `q :  `, consider the language accepted when starting from `q` rather than from `start`. Two states are Nerode equivalent if these "residual" languages are equal:

```
def NerodeEquivalence (s s : ) : Prop :=
  M.acceptsFrom s = M.acceptsFrom s
```

If we quotient the state space by this equivalence relation, we get a DFA whose states are equivalence classes:

```
def toNerodeDFA :
    DFA  (Quotient (M.nerodeEquiv)) where
  step (s' : Quotient (M.nerodeEquiv)) (a : ) :=
    Quotient.lift
      (fun s :   M.step s a)
      (by intros s s h; simp; apply nerodeEquiv.step; apply h) s'
  start := M.start
  accept := {q | q  M.accept }
```

What is happening in the `step` definition: since the next state must be well-defined on equivalence classes, we use `Quotient.lift`. The proof obligation shows that if `s` and `s` are equivalent, then their `step` images are also equivalent, so the definition does not depend on the chosen representative.

6.2 Accessibility and removing unreachable states

Quotienting alone is not enough when the input DFA has unreachable states, because those states still appear as equivalence classes even though they play no role in the language.

We define a state to be **accessible** if it is reachable from the start state by some word:

```
def IsAccessibleState (s : ) : Prop :=
   w : List , M.eval w = s
```

Then we restrict the DFA to the subtype of accessible states. The new state type is `{ s // M.IsAccessibleState s }`, meaning "states paired with a proof they are accessible."

```
def toAccessible : DFA  {s // M.IsAccessibleState s} where
  step s a := M.step s.val a, by
    obtain x, hx := s.prop
    use x ++ [a]
    simp [hx]
  start := M.start, by use []; simp
  accept := {s | s.val  M.accept}
```

Finally, minimization is:

```
def minimize : DFA  (Quotient (M.toAccessible.nerodeEquiv)) :=
  M.toAccessible.toNerodeDFA
```

So `minimize` first removes unreachable states, then collapses Nerode-equivalent states.

7 Computability and finite DFAs

Mathlibs `DFA` definition is intentionally general. If the state type σ is arbitrary and `accept : Set` is an arbitrary predicate, there is no reason acceptance should be decidable or computable.

This project separates two layers:

(1) **Abstract theory**: works for arbitrary σ and `Set` .

(2) **Computable layer**: adds finiteness and decidability assumptions so that we can actually execute constructions like accessibility checking and Nerode equivalence testing.

7.1 A computable accepting set: the `Fin` typeclass

We introduce a small typeclass that stores the accepting states as a `Finset` (a finite, computable set), together with a proof it matches the original `Set` :

```
class Fin (M : DFA  ) where
  finAccept : Finset
  accept_eq : finAccept = M.accept
```

With `Fin M` and `DecidableEq` , membership in the accepting set becomes a decision procedure, so acceptance of a word becomes computable.

8 Decidable Nerode equivalence via bounded equivalence

The definition of Nerode equivalence quantifies over all words (an infinite set), so it is not decidable in general.

The key idea is to reduce the infinite check to a finite one when σ is finite.

8.1 Step 1: bounded Nerode equivalence

For a natural number k, define "k-bounded Nerode equivalence":

> States s_1 and s_2 are k-bounded equivalent if for every word w with $|w| \leq k$, starting from s_1 accepts w iff starting from s_2 accepts w.

This is a finite condition once we know there are only finitely many words up to length k (e.g. when α is finite, or when we restrict to a finite test set produced by an enumeration).

Intuition: k-bounded equivalence compares states only by what they do on short suffixes.

8.2 Step 2: monotonic refinement

As k increases, the bounded equivalence relation can only get finer:

- if two states are distinguishable by some short word, they stay distinguishable
- allowing longer distinguishing words can split equivalence classes, but cannot merge them

So the partition of the state set induced by k-bounded equivalence forms a refinement chain: $k = 0$ is very coarse, then $k = 1$ may split it, and so on.

At $k = 0$, the only word with length ≤ 0 is the empty word `[]`. Therefore, 0-bounded equivalence depends only on whether a state is accepting. This implies there are at most two equivalence classes at $k = 0$ (accepting vs. non-accepting).

8.3 Step 3: stabilization implies full Nerode equivalence

A key lemma is:

> If the k-bounded equivalence relation is the same as the $(k+1)$-bounded relation, then it has stabilized, and it coincides with full Nerode equivalence.

Why this is true (informally): if allowing words of length $k + 1$ does not split any equivalence class, then no longer word can split it either. Any longer distinguishing word would create a split at the first length where the two behaviors diverge, contradicting equality at consecutive steps.

In Lean, this is proved using the DFA transition structure and a "first point of disagreement" argument.

8.4 Step 4: stabilization must happen by $k = |\sigma|$

Now use finiteness of the state set. Each time k increases, exactly one of two things happens:

(a) the partition strictly refines (the number of equivalence classes increases), or

(b) it stabilizes (and then equals full Nerode equivalence by the lemma above)

But the number of equivalence classes can never exceed the number of states $|\sigma|$. Starting from at most 2 classes at $k = 0$, the number of strict refinements is bounded. Therefore, by $k = |\sigma|$ the process must have stabilized, and the k-bounded relation equals full Nerode equivalence.

This yields a decision procedure:

> To test whether s_1 and s_2 are Nerode equivalent, it is enough to test them on all words of length at most $|\sigma|$.

In the project, this is the most technically involved development: we define the bounded relation, prove monotonicity, prove the stabilization lemma, and combine it with the finite pigeonhole argument.

9 Connection to left quotients and `Language.toDFA`

Mathlib also has a construction `Language.toDFA` based on left quotients of a language:

- states are left quotients $w^{-1}L$
- start state is L
- accept states are those quotients containing the empty word

```
def toDFA : DFA  (Set.range L.leftQuotient) where
  step s a := by
    refine s.val.leftQuotient [a], ?_
    obtain y, hy := s.prop
    exists y ++ [a]
    rw [ hy, leftQuotient_append]
  start := L, by exists []
  accept := { s | []  s.val }
```

This starts from an arbitrary language rather than from a DFA, so it does not provide executability or finiteness by itself. In this project we prove that if a DFA `M` recognizes `L`, then `L.toDFA` is equivalent (bijective morphism) to `M.minimize`. This connects the DFA-based construction to the language-theoretic Myhill–Nerode construction already present in Mathlib.

10 Project structure

- `MyProject/DFA/Hom.lean` — morphisms, surjections, equivalences, preorder
- `MyProject/DFA/Accessible.lean` — accessibility, `toAccessible`, decidability infrastructure
- `MyProject/DFA/Nerode.lean` — Nerode equivalence, bounded equivalence, stabilization results
- `MyProject/DFA/Minimize.lean` — minimization construction and correctness, uniqueness up to equivalence
- `MyProject/DFA/Fin.lean` — computable DFAs via `Fin` accept sets

11 Current developments

I initially intended this as a Mathlib contribution and discussed it on Lean Zulip. I was pointed to the community project **CSLib**, which targets computer science formalizations more directly. CSLib uses different DFA definitions, but currently also lacks morphisms, accessibility, and minimization. I am refactoring this development to CSLibs definitions with the goal of contributing it there.

`https://leanprover.zulipchat.com/#narrow/channel/287929-mathlib4/topic/Automata.20The`

12 TODO

- Add textbook references (Hopcroft and Ullman, and standard Myhill–Nerode presentations).
- Add a short note comparing this construction with Hopcrofts algorithm, and why the quotient construction is the right choice for a proof-oriented formalization.