A Formalisation of Finite Automata using Hereditarily Finite Sets

Lawrence C. Paulson

Computer Laboratory, University of Cambridge, England
lp15@cam.ac.uk

Abstract. Hereditarily finite (HF) set theory provides a standard universe of sets, but with no infinite sets. Its utility is demonstrated through a formalisation of the theory of regular languages and finite automata, including the Myhill-Nerode theorem and Brzozowski's minimisation algorithm. The states of an automaton are HF sets, possibly constructed by product, sum, powerset and similar operations.

1 Introduction

The theory of finite state machines is fundamental to computer science. It has applications to lexical analysis, hardware design and regular expression pattern matching. A regular language is one accepted by a finite state machine, or equivalently, one generated by a regular expression or a type-3 grammar [6]. Researchers have been formalising this theory for nearly three decades.

A critical question is how to represent the states of a machine. Automata theory is developed using set-theoretic constructions, e.g. the product, disjoint sum or powerset of sets of states. But in a strongly-typed formalism such as higher-order logic (HOL), machines cannot be polymorphic in the type of states: statements such as "every regular language is accepted by a finite state machine" would require existential quantification over types. One might conclude that there is no good way to formalise automata in HOL [5, 15].

This paper sets out to show that finite automata theory can be formalised within the theory of *hereditarily finite sets*: set theory with the negation of the axiom of infinity. It admits the usual constructions, including lists, functions and integers, but no infinite sets. The type of HF sets can be constructed from the natural numbers within higher-order logic. We could just use numbers, but with ugly coding everywhere. Using HF sets, we retain the textbook definitions.

The paper introduces HF set theory and automata (Sect. 2). It presents a formalisation of deterministic finite automata and results such as the Myhill-Nerode theorem (Sect. 3). It also treats nondeterministic finite automata and results such as the powerset construction and closure under regular expression operations (Sect. 4). Next come minimal automata, their uniqueness up to isomorphism, and Brzozowski's algorithm for minimising an automaton [3] (Sect. 5). The paper concludes after discussing related work (Sect. 6–7). The proofs, which are available online [12], also demonstrate the use of Isabelle's *locales* [1].

2 Background

An *hereditarily finite set* can be understood inductively as a finite set of hereditarily finite sets [14]. This definition justifies the recursive definition $f(x) = \sum \{2^{f(y)} \mid y \in x\}$, yielding a bijection $f : \mathrm{HF} \to \mathbb{N}$ between the HF sets and the natural numbers. The linear ordering on HF given by $x < y \iff f(x) < f(y)$ can be shown to extend both the membership and the subset relations.

The HF sets support many standard constructions, even quotients. Equivalence classes are not available in general — they may be infinite — but the linear ordering over HF identifies a unique representative. The integers and rationals can be constructed, with their operations (but not the *set* of integers, obviously). Świerczkowski [14] has used HF as the basis for proving Gödel's incompleteness theorems, and I have formalised his work using Isabelle [13].

Let Σ be a nonempty, finite alphabet of *symbols*. Then Σ^* is the set of *words*: finite sequences of symbols. The empty word is written ϵ, and the concatenation of words u and v is written uv. A *deterministic finite automaton* (DFA) [6, 7] is a structure $(K, \Sigma, \delta, q_0, F)$ where K is a finite set of states, $\delta : K \times \Sigma \to K$ is the next-state function, $q_0 \in K$ is the initial state and $F \subseteq K$ is the set of final or accepting states. The next-state function on symbols is extended to one on words, $\delta^* : K \times \Sigma^* \to K$ such that $\delta^*(q, \epsilon) = q$, $\delta^*(q, a) = \delta(q, a)$ for $a \in \Sigma$ and $\delta^*(q, uv) = \delta^*(\delta^*(q, u), v)$. The DFA *accepts* the string w if $\delta^*(q_0, w) \in F$. A set $L \subseteq \Sigma^*$ is a *regular language* if L is the set of strings accepted by some DFA.

A *nondeterministic finite automaton* (NFA) is similar, but admits multiple execution paths and accepts a string if one of them reaches a final state. Formally, an NFA is a structure $(K, \Sigma, \delta, Q_0, F)$ where $\delta : K \times \Sigma \to \mathcal{P}(K)$ is the next-state function, $Q_0 \subseteq K$ a set of initial states, the other components as above. The next-state function is extended to $\delta^* : \mathcal{P}(K) \times \Sigma^* \to \mathcal{P}(K)$ such that $\delta^*(Q, \epsilon) = Q$, $\delta^*(Q, a) = \bigcup_{q \in Q} \delta(q, a)$ for $a \in \Sigma$ and $\delta^*(Q, uv) = \delta^*(\delta^*(Q, u), v)$. An NFA accepts the string w provided $\delta^*(q, w) \in F$ for some $q \in Q_0$.

The notion of NFA can be extended with ϵ-transitions, allowing "silent" transitions between states. Define the transition relation $q \xrightarrow{a} q'$ for $q' \in \delta(q, a)$. Let the ϵ-transition relation $q \xrightarrow{\epsilon} q'$ be given. Then define the transition relation $q \xRightarrow{a} q'$ to allow ϵ-transitions before and after: $(\xrightarrow{\epsilon})^* \circ (\xrightarrow{a}) \circ (\xrightarrow{\epsilon})^*$.

Every NFA can be transformed into a DFA, where the set of states is the powerset of the NFA's states, and the next-state function captures the effect of $q \xRightarrow{a} q'$ on these sets of states. Thus, we can use DFAs and NFAs interchangeably. Regular languages are easily shown to be closed under intersection and complement, therefore also under union. They are closed under repetition (the Kleene-star operation). Two especially important results are discussed below:

– The Myhill-Nerode theorem gives necessary and sufficient conditions for a language to be regular. It defines a canonical and minimal DFA for any given regular language. Minimal DFAs are unique up to isomorphism.
– Reorienting the arrows of the transition relation transforms a DFA into an NFA accepting the reverse of the given language. We can regain a DFA using

the powerset construction. Repeating this operation yields a minimal DFA for the original language. This is Brzozowski's minimisation algorithm [3].

This work has been done using the proof assistant Isabelle/HOL. Documentation is available online at `http://isabelle.in.tum.de/`. The work refers to equivalence relations and equivalence classes, following the conventions established in my earlier paper [11]. If `R` is an equivalence relation on the set `A`, then `A//R` is the set of equivalence classes. If $x \in A$, then its equivalence class is `R''{x}`. Formally, it is the image of `x` under `R`: the set of all `y` such that $(x,y) \in R$. More generally, if $X \subseteq A$ then `R''X` is the union of the equivalence classes `R''{x}` for $x \in X$.

3   Deterministic Automata; the Myhill-Nerode Theorem

When adopting HF set theory, there is the question of whether to use it for everything, or only where necessary. The set of states is finite, so it could be an HF set, and similarly for the set of final states. The alphabet could also be given by an HF set; then words—lists of symbols—would also be HF sets. Our definitions could be essentially typeless.

The approach adopted here is less radical. It makes a minimal use of HF, taking the maximum advantage of types. Standard HOL sets (which are effectively predicates) are intermixed with HF sets. Definitions are polymorphic in the type `'a` of alphabet symbols, while words have type `'a list`.

3.1   Basic Definition of DFAs

The record definition below declares the components of a DFA. The types make it clear that there is indeed a set of states but only a single initial state, etc.

```
record 'a dfa = states :: "hf set"
                init   :: "hf"
                final  :: "hf set"
                nxt    :: "hf ⇒ 'a ⇒ hf"
```

Once we have decided upon the representation of the DFA, the natural approach in Isabelle is to package up the axioms of the DFA as a locale [1]:

```
locale dfa =
  fixes M :: "'a dfa"
  assumes init:  "init M ∈ states M"
      and final: "final M ⊆ states M"
      and nxt:   "⋀q x. q ∈ states M ⟹ nxt M q x ∈ states M"
      and finite: "finite (states M)"
```

The assumptions listed above form a local context, and are available within the locale. It is then easy to define the accepted language. Note that `language` is not a function but a *constant* within the locale, referring to one particular DFA.

```
primrec nextl :: "hf ⇒ 'a list ⇒ hf" where
    "nextl q []    = q"
  | "nextl q (x#xs) = nextl (nxt M q x) xs"
```

definition `language :: "('a list) set"` **where**
  `"language ≡ {xs. nextl (init M) xs ∈ final M}"`

Equivalence relations play a significant role below. The following relation regards two strings as equivalent if they take the machine to the same state [7, p. 90].

definition `eq_nextl :: "('a list × 'a list) set"` **where**
  `"eq_nextl ≡ {(u,v). nextl (init M) u = nextl (init M) v}"`

3.2 Myhill-Nerode Relations

The Myhill-Nerode theorem asserts the equivalence of three characterisations of regular languages. The first of these is to be the language accepted by some DFA. The other two are connected with certain equivalence relations, called Myhill-Nerode relations, on words of the language.

The definitions below are outside of the locale and are therefore independent of any particular DFA. The predicate `dfa` refers to the locale axioms and expresses that its argument, `M`, is a DFA. The predicate `dfa.language` refers to the constant `language`: outside of the locale, it takes a DFA as an argument.

definition `regular :: "('a list) set ⇒ bool"` **where**
  `"regular L ≡ ∃M. dfa M ∧ dfa.language M = L"`

The other characterisations of a regular language involve abstract finite state machines derived from the language itself, with certain equivalence classes as the states. A relation is *right invariant* if it satisfies the following closure property.

definition `right_invariant :: "('a list × 'a list) set ⇒ bool"` **where**
  `"right_invariant r ≡ (∀u v w. (u,v) ∈ r ⟶ (u@w, v@w) ∈ r)"`

The intuition is that if two words `u` and `v` are related, then each word brings the "machine" to the same state, and once this has happened, this agreement must continue no matter how the words are extended as `u@w` and `v@w`.

A *Myhill-Nerode relation* for a language `L` is a right invariant equivalence relation of finite index where `L` is the union of some of the equivalence classes [7, p. 90]. *Finite index* means the set of equivalence classes is finite: `finite (UNIV//R)`. The equivalence classes will be the states of a finite state machine. The equality `L = R''A`, where `A ⊆ L` is a set of words of the language, expresses `L` as the union of a set of equivalence classes, which will be the final states.

definition `MyhillNerode :: "'a list set ⇒ ('a list * 'a list)set ⇒ bool"`
  where `"MyhillNerode L R ≡ equiv UNIV R ∧ right_invariant R ∧`
                     `finite (UNIV//R) ∧ (∃A. L = R''A)"`

While `eq_nextl` (defined in §3.1) refers to a machine, the relation `eq_app_right` is defined in terms of a language, `L`. It relates the words `u` and `v` if all extensions of them, `u@w` and `v@w`, behave equally with respect to `L`:

definition `eq_app_right :: "'a list set ⇒ ('a list * 'a list) set"` **where**
  `"eq_app_right L ≡ {(u,v). ∀w. u@w ∈ L ⟷ v@w ∈ L}"`

It is a Myhill-Nerode relation for `L` provided it is of finite index:

lemma `MN_eq_app_right:`
    `"finite (UNIV // eq_app_right L) ⟹ MyhillNerode L (eq_app_right L)"`

Moreover, every Myhill-Nerode relation `R` for `L` refines `eq_app_right L`.

lemma `MN_refines_eq_app_right: "MyhillNerode L R ⟹ R ⊆ eq_app_right L"`

This essentially states that `eq_app_right L` is the most abstract Myhill-Nerode relation for `L`. This will eventually yield a way of defining a minimal machine.

3.3 The Myhill-Nerode Theorem

The Myhill-Nerode theorem says that these three statements are equivalent [6]:

1. The set `L` is a regular language (is accepted by some DFA).
2. There exists some Myhill-Nerode relation `R` for `L`.
3. The relation `eq_app_right L` has finite index.

We have (1) ⇒ (2) because `eq_nextl` is a Myhill-Nerode relation. We have (2) ⇒ (3), by lemma `MN_refines_eq_app_right`, because every equivalence class for `eq_app_right L` is the union of equivalence classes of `R`, and so `eq_app_right L` has minimal index for all Myhill-Nerode relations. We get (3) ⇒ (1) by constructing a DFA whose states are the (finitely many) equivalence classes of `eq_app_right L`. This construction can be done for every Myhill-Nerode relation.

   Until now, all proofs have been routine. But now we face a difficulty: the states of our machine should be equivalence classes of words, but these could be infinite sets. What can be done? The solution adopted here is to map the equivalence classes to the natural numbers, which are easily embedded in HF.

   This solution is simple but not appealing, and may suggest that there is no advantage in using the HF sets instead of the natural numbers. However, mapping sets to integers turns out to be convenient only occasionally, and even here, it is not necessary. If symbols are HF sets, then so are words; we could represent the states of the DFA by representatives chosen from the equivalence classes. We can do this even while retaining the polymorphism in type `'a dfa`: via Isabelle's type-class system, it is easy to define the class of types that can be embedded into HF, including the integers, booleans, lists of such, etc.

3.4 Constructing a DFA from a Myhill-Nerode Relation

If `R` is a Myhill-Nerode relation for a language `L`, then the set of equivalence classes is finite, and a subset of them determine `L`. These things describe a DFA for `L`. The construction is packaged as a locale, which is used once in the proof of the Myhill-Nerode theorem, and again to prove that minimal DFAs are unique.

The locale includes not only `L` and `R`, but also the set `A` of accepting states, the cardinality `n` and the bijection `h` between the set `UNIV//R` of equivalence classes and the number `n` as represented in HF. The locale assumes the Myhill-Nerode conditions.

```
locale MyhillNerode_dfa =
  fixes L :: "('a list) set" and R :: "('a list * 'a list) set"
   and A :: "('a list) set" and n :: nat and h :: "('a list) set ⇒ hf"
  assumes eqR: "equiv UNIV R"
      and riR: "right_invariant R"
      and L:   "L = R''A"
      and h:   "bij_betw h (UNIV//R) (hfset (ord_of n))"
```

The DFA is defined within the locale. The states are given by the equivalence classes. The initial state is the equivalence class for the empty word; the set of final states is derived from the set `A` of words that generate `L`; the next-state function maps the equivalence class for the word `u` to that for `u@[x]`. Equivalence classes are not the actual states here, but are mapped to integers via the bijection `h`. As mentioned above, this use of integers is not essential.

```
  definition DFA :: "'a dfa" where
    "DFA = (|states = h ' (UNIV//R),
            init   = h (R '' {[]}),
            final  = {h (R '' {u}) | u. u ∈ A},
            nxt    = λq x. h (⋃u ∈ h⁻¹ q. R '' {u@[x]})|)"
```

This can be proved to be a DFA easily. One proof line, using the right-invariance property and lemmas about quotients [11], proves that the next-state function respects the equivalence relation. Four more lines are needed to verify the properties of a DFA, somewhat more to show that the language of this DFA is indeed `L`.

   The facts proved within the locale are summarised (outside its scope) by the following theorem, stating that every Myhill-Nerode relation yields an equivalent DFA. (The **obtains** form expresses existential and multiple conclusions.)

```
theorem MN_imp_dfa:
  assumes "MyhillNerode L R"
  obtains M where "dfa M"  "dfa.language M = L"
                  "card (states M) = card (UNIV//R)"
```

This completes the (3) ⇒ (1) stage, by far the hardest, of the Myhill-Nerode theorem. The three stages are shown below. Lemma `L2_3` includes a result about cardinality: the construction yields a minimal DFA, which will be useful later.

```
lemma L1_2: "regular L ⟹ ∃R. MyhillNerode L R"
lemma L2_3:
  assumes "MyhillNerode L R"
  obtains "finite (UNIV // eq_app_right L)"
          "card (UNIV // eq_app_right L) ≤ card (UNIV // R)"
lemma L3_1: "finite (UNIV // eq_app_right L) ⟹ regular L"
```

4 Nondeterministic Automata and Closure Proofs

As most of the proofs are simple, our focus will be the use of HF sets when defining automata. Our main example is the powerset construction for transforming a nondeterministic automaton into a deterministic one.

4.1 Basic Definition of NFAs

As in the deterministic case, a record holds the necessary components, while a locale encapsulates the axioms. Component `eps` deals with ϵ-transitions.

```
record 'a nfa = states :: "hf set"
                init   :: "hf set"
                final  :: "hf set"
                nxt    :: "hf ⇒ 'a ⇒ hf set"
                eps    :: "(hf * hf) set"
```

The axioms are obvious: the initial, final and next states belong to the set of states, which is finite. An axiom restricting ϵ-transitions to machine states was removed: it did not simplify proofs. Working with ϵ-transitions is messy. Allowing multiple initial states instead of one reduces the need for ϵ-transitions. It also helps to provide special treatment for NFAs having no ϵ-transitions.

```
locale nfa =
  fixes M :: "'a nfa"
  assumes init: "init M ⊆ states M"
      and final: "final M ⊆ states M"
      and nxt:   "⋀q x. q ∈ states M ⟹ nxt M q x ⊆ states M"
      and finite: "finite (states M)"
```

The following function "closes up" a set `Q` of states under ϵ-transitions. Intersection with `states M` confines these transitions to legal states.

```
definition epsclo :: "hf set ⇒ hf set" where
  "epsclo Q ≡ states M ∩ (⋃q∈Q. {q'. (q,q') ∈ (eps M)*})"
```

The remaining definitions are straightforward. Note that `nextl` generalises `nxt` to take a set of states as well is a list of symbols.

```
primrec nextl :: "hf set ⇒ 'a list ⇒ hf set" where
    "nextl Q []     = epsclo Q"
  | "nextl Q (x#xs) = nextl (⋃q ∈ epsclo Q. nxt M q x) xs"
definition language :: "('a list) set"  where
  "language ≡ {xs. nextl (init M) xs ∩ final M ≠ {}}"
```

4.2 The Powerset Construction

The construction of a DFA to simulate a given NFA is elementary, and is a good demonstration of the HF sets. The strongly-typed approach used here requires a pair of coercion functions `hfset :: "hf ⇒ hf set"` and `HF :: "hf set ⇒ hf"` to convert between HF sets and ordinary sets.

```
lemma HF_hfset: "HF (hfset a) = a"
lemma hfset_HF: "finite A ⟹ hfset (HF A) = A"
```

With this approach, type-checking indicates whether we are dealing with a set of states or a single state. The drawback is that we occasionally have to show that a set of states is finite in the course of reasoning about the coercions, which would never be necessary if we confined our reasoning to the HF world.

Here is the definition of the DFA. The states are ϵ-closed subsets of NFA states, coerced to type `hf`. The initial and final states are defined similarly, while the next-state function requires both coercions and performs ϵ-closure before and after. We work in locale `nfa`, with access to the components of the NFA.

```
definition Power_dfa :: "'a dfa" where
 "Power_dfa = (|dfa.states = HF ` epsclo ` Pow (states M),
          init  = HF(epsclo(init M)),
          final = {HF(epsclo Q) | Q. Q ⊆ states M ∧ Q ∩ final M ≠ {}},
          nxt   = λQ x. HF(⋃ q ∈ epsclo (hfset Q). epsclo (nxt M q x))|)"
```

Proving that this is a DFA is trivial. The hardest case is to show that the next-state function maps states to states. Proving that the two automata accept the same language is also simple, by reverse induction on lists (the induction step concerns `u@[x]`, putting `x` at the end). Here, `Power.language` refers to the language of the powerset DFA, while `language` refers to that of the NFA.

```
theorem Power_language: "Power.language = language"
```

4.3   Other Closure Properties

The set of languages accepted by some DFA is closed under complement, intersection, concatenation, repetition (Kleene star), etc. [6]. Consider intersection:

```
theorem regular_Int:
  assumes S: "regular S" and T: "regular T" shows "regular (S ∩ T)"
```

The recognising DFA is created by forming the Cartesian product of the sets of states of `MS` and `MT`, the DFAs of the two languages. The machines are effectively run in parallel. The decision to represent a set of states by type `hf set` rather than by type `hf` means we cannot write `dfa.states MS × dfa.states MT`, but we can express this concept using set comprehension:

```
 "(|states = {⟨q1,q2⟩ | q1 q2. q1 ∈ dfa.states MS ∧ q2 ∈ dfa.states MT},
   init  = ⟨dfa.init MS, dfa.init MT⟩,
   final = {⟨q1,q2⟩ | q1 q2. q1 ∈ dfa.final MS ∧ q2 ∈ dfa.final MT},
   nxt   = λ⟨qs,qt⟩ x. ⟨dfa.nxt MS qs x, dfa.nxt MT qt x⟩|)"
```

This is trivially shown to be a DFA. Showing that it accepts the intersection of the given languages is again easy by reverse induction.

Closure under concatenation is expressed as follows:

```
theorem regular_conc:
  assumes S: "regular S" and T: "regular T" shows "regular (S @@ T)"
```

8

The concatenation is recognised by an NFA involving the disjoint sum of the sets of states of `MS` and `MT`, the DFAs of the two languages. The effect is to simulate the first machine until it accepts a string, then to transition to a simulation of the second machine. There are ϵ-transitions linking every final state of `MS` to the initial state of `MT`. We again cannot write `dfa.states MS + dfa.states MT`, but we can express the disjoint sum naturally enough:

```
"(|states = Inl ' (dfa.states MS) ∪ Inr ' (dfa.states MT),
   init  = {Inl (dfa.init MS)},
   final = Inr ' (dfa.final MT),
   nxt   = λq x. sum_case (λqs. {Inl (dfa.nxt MS qs x)})
                          (λqt. {Inr (dfa.nxt MT qt x)}) q,
   eps   = (λq. (Inl q, Inr (dfa.init MT))) ' dfa.final MS|)"
```

Again, it is trivial to show that this is an NFA. But unusually, proving that it recognises the concatenation of the languages is a challenge. We need to show, by induction, that the "left part" of the NFA correctly simulates `MS`.

```
have "⋀q. Inl q ∈ ST.nextl {Inl (dfa.init MS)} u ⟷
         q = (dfa.nextl MS (dfa.init MS) u)"
```

The key property is that any string accepted by the NFA can be split into strings accepted by the two DFAs. The proof involves a fairly messy induction.

```
have "⋀q. Inr q ∈ ST.nextl {Inl (dfa.init MS)} u ⟷
         (∃uS uT. uS ∈ dfa.language MS ∧ u = uS@uT ∧
                  q = dfa.nextl MT (dfa.init MT) uT)"
```

Closure under Kleene star is not presented here, as it involves no interesting set operations. The language L^* is recognised by an NFA with an extra state, which serves as the initial state and runs the DFA for L including iteration. The proofs are messy, with many cases. To their credit, Hopcroft and Ullman [6] give some details, while other authors content themselves with diagrams alone.

5 State Minimisation for DFAs

The Myhill-Nerode theorem describes a DFA for a regular language L having the minimum number of states. But it does not yield a minimisation algorithm. It turns out that a DFA is minimal if it has no unreachable states and if no two states are *indistinguishable* (in a sense made precise below). This again does not yield an algorithm. *Brzozowski's minimisation algorithm* involves reversing the DFA to create an NFA, converting back to a DFA via powersets, removing unreachable states, then repeating those steps to undo the reversal. Surprisingly, it performs well in practice [3].

5.1 The Left and Right Languages of a State

The following developments are done within the locale `dfa`, and therefore refer to one particular deterministic finite automaton.

The *left language* of a state q is the set of all words w such that $q_0 \xrightarrow{w}^* q$, or informally, such that the machine when started in the initial state and given the word w ends up in q. In a DFA, the left languages of distinct states are disjoint, if they are nonempty.

definition `left_lang :: "hf ⇒ ('a list) set"` **where**
  `"left_lang q ≡ {u. nextl (init M) u = q}"`

The *right language* of a state q is the set of all words w such that $q \xrightarrow{w}^* q_f$, where q_f is a final state, or informally, such that the machine when started in q will accept the word w. The language of a DFA is the right language of q_0. Two states having the same right language are *indistinguishable*: they both lead to the same words being accepted.

definition `right_lang :: "hf ⇒ ('a list) set"` **where**
  `"right_lang q ≡ {u. nextl q u ∈ final M}"`

The *accessible* states are those that can be reached by at least one word.

definition `accessible :: "hf set"` **where**
  `"accessible ≡ {q. left_lang q ≠ {}}"`

The function `path_to` returns one specific such word. This function will eventually be used to express an isomorphism between any minimal DFA (one having no inaccessible or indistinguishable states) and the canonical DFA determined by the Myhill-Nerode theorem.

definition `path_to :: "hf ⇒ 'a list"` **where**
  `"path_to q ≡ SOME u. u ∈ left_lang q"`
lemma `nextl_path_to:`
  `"q ∈ accessible ⟹ nextl (dfa.init M) (path_to q) = q"`

First, we deal with the problem of inaccessible states. It is easy to restrict any DFA to one having only accessible states.

definition `Accessible_dfa :: "'a dfa"` **where**
  `"Accessible_dfa = (|dfa.states = accessible,`
  `                    init  = init M,`
  `                    final = final M ∩ accessible,`
  `                    nxt   = nxt M|)"`

This construction is readily shown to be a DFA that agrees with the original in most respects. In particular, the two automata agree on `left_lang` and `right_lang`, and therefore on the language they accept:

lemma `Accessible_language: "Accessible.language = language"`

We can now define a DFA to be minimal if all states are accessible and no two states have the same right language. (The formula `inj_on right_lang (dfa.states M)` expresses that the function `right_lang` is injective on the set `dfa.states M`.)

definition `minimal` **where**
  `"minimal ≡ accessible = states M ∧ inj_on right_lang (dfa.states M)"`

Because we are working within the DFA locale, `minimal` is a constant referring
to one particular automaton.

5.2 A Collapsing Construction

We can deal with indistinguishable states similarly, defining a DFA in which the
indistinguishable states are identified via equivalence classes. This is not part
of Brzozowski's minimisation algorithm, but it is interesting in its own right:
the equivalence classes themselves are HF sets. We begin by declaring a relation
stating that two states are equivalent if they have the same right language.

definition `eq_right_lang :: "(hf × hf) set"` **where**
  `"eq_right_lang ≡ {(u,v). u ∈ states M ∧ v ∈ states M ∧`
                           `right_lang u = right_lang v}"`

Trivially, this is an equivalence relation, and equivalence classes of states are
finite (there are only finitely many states). In the corresponding DFA, these
equivalence classes form the states, with the initial and final states given by the
equivalence classes for the corresponding states of the original DFA. As usual,
the function `HF` is used to coerce a set of states to type `hf`.

definition `Collapse_dfa :: "'a dfa"` **where**
  `"Collapse_dfa = (|dfa.states = HF ' (states M // eq_right_lang),`
                  `init  = HF (eq_right_lang '' {init M}),`
                  `final = {HF (eq_right_lang '' {q}) | q. q ∈ final M},`
          `nxt = λQ x. HF (⋃q ∈ hfset Q. eq_right_lang '' {nxt M q x})|)"`

This is easily shown to be a DFA, and the next-state function respects the equiv-
alence relation. Showing that it accepts the same language is straightforward.

lemma `ext_language_Collapse_dfa:`
    `"u ∈ Collapse.language ⟷ u ∈ language"`

5.3 The Uniqueness of Minimal DFAs

The property `minimal` is true for machines having no inaccessible or indistin-
guishable states. To prove that such a machine actually has a minimal number
of states is tricky. It can be shown to be isomorphic to the canonical machine
from the Myhill-Nerode theorem, which indeed has a minimal number of states.
    Automata `M` and `N` are *isomorphic* if there exists a bijection `h` between their
state sets that preserves their initial, final and next states. This conception is
nicely captured by a locale, taking the DFAs as parameters:

locale `dfa_isomorphism = M: dfa M + N: dfa N`
                      for `M :: "'a dfa"` **and** `N :: "'a dfa"` +
  fixes `h :: "hf ⇒ hf"`

11

```
assumes h: "bij_betw h (states M) (states N)"
    and init : "h (init M) = init N"
    and final: "h ' final M = final N"
    and nxt  : "⋀q x. q ∈ states M ⟹ h(nxt M q x) = nxt N (h q) x"
```

With this concept at our disposal, we resume working within the locale `dfa`, which is concerned with the DFA `M`. Under the assumption that no two states have the same right language, there is a bijection between the accessible states (of `M`) and the equivalence classes yielded by the relation `eq_app_right language`.

```
lemma inj_right_lang_imp_eq_app_right_index:
  assumes "inj_on right_lang (dfa.states M)"
    shows "bij_betw (λq. eq_app_right language '' {path_to q})
                    accessible  (UNIV // eq_app_right language)"
```

This bijection maps the state `q` to `eq_app_right language '' {path_to q}`. Every element of the quotient `UNIV // eq_app_right language` can be expressed in this form. And therefore, the number of states in a `minimal` machine equals the index of `eq_app_right language`.

```
definition min_states where
  "min_states ≡ card (UNIV // eq_app_right language)"
lemma minimal_imp_index_eq_app_right:
  "minimal ⟹ card(dfa.states M) = min_states"
```

In the proof of the Myhill-Nerode theorem, it emerged that this index was the minimum cardinality for any DFA accepting the given language. Any other automaton, `M'`, accepting the same language cannot have fewer states. This theorem justifies the claim that `minimal` indeed characterises a minimal DFA.

```
theorem minimal_imp_card_states_le:
    "⟦minimal; dfa M'; dfa.language M' = language⟧
    ⟹ card (dfa.states M) ≤ card (dfa.states M')"
```

Note that while the locale `dfa` gives us implicit access to one DFA, namely `M`, it is still possible to refer to other automata, as we see above.

The minimal machine is unique up to isomorphism because every minimal machine is isomorphic to the canonical Myhill-Nerode DFA. The construction of a DFA from a Myhill-Nerode relation was packaged as a locale, and by applying this locale to the given `language` and the relation `eq_app_right language`, we can generate the instance we need.

```
interpretation Canon:
    MyhillNerode_dfa language "eq_app_right language"
                      language min_states index_f
```

Here, `index_f` denotes some bijection between the equivalence classes and their cardinality (as an HF ordinal). It exists (definition omitted) by the definition of cardinality itself. It is the required isomorphism function between `M` and the canonical DFA of Sect. 3.4, which is written `Canon.DFA`.

definition `iso :: "hf ⇒ hf"` **where**
  `"iso ≡ index_f o (λq. eq_app_right language `` {path_to q})"`

The isomorphism property is stated using locale `dfa_isomorphism`.

theorem `minimal_imp_isomorphic_to_canonical:`
  assumes `minimal`   **shows** `"dfa_isomorphism M Canon.DFA iso"`

Verifying the isomorphism conditions requires delicate reasoning. Hopcroft and Ullman's proof [6, p. 29–30] provides just a few clues.

5.4 Brzozowski's Minimisation Algorithm

At the core of this minimisation algorithm is an NFA obtained by reversing all the transitions of a given DFA, and exchanging the initial and final states.

definition `Reverse_nfa :: "'a dfa ⇒ 'a nfa"` **where**
  `"Reverse_nfa MS = (|nfa.states = dfa.states MS,`
  `                     init  = dfa.final MS,`
  `                     final = {dfa.init MS},`
  `                     nxt = λq x. {p ∈ dfa.states MS. q = dfa.nxt MS p x},`
  `                     eps = {}|)"`

This is easily shown to be an NFA that accepts the reverse of every word accepted by the original DFA. Applying the powerset construction yields a new DFA that has no indistinguishable states. The point is that the right language of a powerset state is derived from the right languages of the constituent states of the reversal NFA [3]. Those, in turn, are the left languages of the original DFA, and these are disjoint (since the original DFA has no inaccessible states, by assumption).

lemma `inj_on_right_lang_PR:`
  assumes `"dfa.states M = accessible"`
    shows `"inj_on (dfa.right_lang (nfa.Power_dfa (Reverse_nfa M)))`
  `                (dfa.states (nfa.Power_dfa (Reverse_nfa M)))"`

The following definitions abbreviate the steps of Brzozowski's algorithm.

abbreviation `APR :: "'x dfa ⇒ 'x dfa"` **where**
  `"APR X ≡ dfa.Accessible_dfa (nfa.Power_dfa (Reverse_nfa X))"`
definition `Brzozowski :: "'a dfa"` **where**
  `"Brzozowski ≡ APR (APR M)"`

  By the lemma proved just above, the `APR` operation yields minimal DFAs.

theorem `minimal_APR:`
  assumes `"dfa.states M = accessible"`
    shows `"dfa.minimal (APR M)"`

Brzozowski's minimisation algorithm is correct. The first APR call reverses the language and eliminates inaccessible states; the second call yields a minimal machine for the original language. The proof uses the theorems just proved.

```
theorem minimal_Brzozowski: "dfa.minimal Brzozowski"
unfolding Brzozowski_def
proof (rule dfa.minimal_APR)
  show "dfa (APR M)"
    by (simp add: dfa.dfa_Accessible nfa.dfa_Power nfa_Reverse_nfa)
next
  show "dfa.states (APR M) = dfa.accessible (APR M)"
    by (simp add: dfa.Accessible_accessible dfa.states_Accessible_dfa
            nfa.dfa_Power nfa_Reverse_nfa)
qed
```

6   Related Work

There is a great body of prior work. One approach involves working constructively, in some sort of type theory. Constable's group has formalised automata [4] in Nuprl, including the Myhill-Nerode theorem. Using type theory in the form of Coq and its Ssreflect library, Doczkal et al. [5] formalise much of the same material as the present paper. They omit ϵ-transitions and Brzozowski's algorithm and add the pumping lemma and Kleene's algorithm for translating a DFA to a regular expression. Their development is of a similar length, under 1400 lines, and they allow the states of a finite automaton to be given by any finite type. In a substantial development, Braibant and Pous [2] have implemented a tactic for solving equations in Kleene algebras by implementing efficient finite automata algorithms in Coq. They represent states by integers.

An early example of regular expression theory formalised using higher-order logic (Isabelle/HOL) is Nipkow's verified lexical analyser [9]. His automata are polymorphic in the types of state and symbols. NFAs are included, with ϵ-transitions simulated by an alphabet extended with a dummy symbol.

Recent Isabelle developments explicitly bypass automata theory. Wu et al. [15] prove the Myhill-Nerode theorem using regular expressions. This is a significant feat, especially considering that the theorem's underlying intuitions come from automata. Current work on regular expression equivalence [8, 10] continues to focus on regular expressions rather than finite automata.

It is no criticism to say that finite automata cause pervasive issues. Constructive logic will not appeal to everybody, and quotient types can cause complications. The HF sets make things simple. This paper describes not a project undertaken by a team, but a six-week case study by one person.

7   Conclusions

The theory of finite automata can be developed straightforwardly using higher-order logic and hereditarily finite set theory. We can formalise the textbook proofs: there is no need to avoid automata or use constructive type theories. HF set theory can be seen as an abstract universe of computable objects. It could have many other applications. One possibility is programming language

semantics: using `hf` as the type of values offers open-ended possibilities, including integer, rational and floating point numbers, ASCII characters, and data structures built up from them.

References

1. Clemens Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014.
2. Thomas Braibant and Damien Pous. Deciding Kleene algebras in Coq. *Logical Methods in Computer Science*, 8(1), 2012.
3. Jean-Marc Champarnaud, Ahmed Khorsi, and Thomas Paranthoën. Split and join for minimizing: Brzozowski's algorithm. In Miroslav Balík and Milan Simánek, editors, *The Prague Stringology Conference*, pages 96–104. Department of Computer Science and Engineering, Czech Technical University, 2002.
4. Robert L. Constable, Paul B. Jackson, Pavel Naumov, and Juan C. Uribe. Constructively formalizing automata theory. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 213–238. MIT Press, 2000.
5. Christian Doczkal, Jan-Oliver Kaiser, and Gert Smolka. A constructive theory of regular languages in Coq. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, LNCS 8307, pages 82–97. Springer, 2013.
6. John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.
7. Dexter Kozen. *Automata and computability*. Springer, New York, 1997.
8. Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Autom. Reasoning*, 49(1):95–106, 2012.
9. Tobias Nipkow. Verified lexical analysis. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: TPHOLs '98*, LNCS 1479, pages 1–15. Springer, 1998. Invited lecture.
10. Tobias Nipkow and Dmitriy Traytel. Unified decision procedures for regular expression equivalence. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving — 5th International Conference, ITP 2014*, LNCS 8558, pages 450–466. Springer, 2014.
11. Lawrence C. Paulson. Defining functions on equivalence classes. *ACM Transactions on Computational Logic*, 7(4):658–675, 2006.
12. Lawrence C. Paulson. Finite automata in hereditarily finite set theory. *Archive of Formal Proofs*, February 2015. `http://afp.sf.net/entries/Finite_Automata_HF.shtml`, Formal proof development.
13. Lawrence C. Paulson. A mechanised proof of Gödel's incompleteness theorems using Nominal Isabelle. *Journal of Automated Reasoning*, 2015. In press.
14. S. Świerczkowski. Finite sets and Gödel's incompleteness theorems. *Dissertationes Mathematicae*, 422:1–58, 2003. `http://journals.impan.gov.pl/dm/Inf/422-0-1.html`.
15. Chunhan Wu, Xingyuan Zhang, and Christian Urban. A formalisation of the Myhill-Nerode theorem based on regular expressions. *Journal of Automated Reasoning*, 52(4):451–480, 2014.