Finite Automata using the Hereditarily Finite Sets

Prof. Lawrence C Paulson
Computer Laboratory, University of Cambridge

Abstract

Finite Automata, both deterministic and non-deterministic, for regular languages. The Myhill-Nerode Theorem. Closure under intersection, concatenation, etc. Regular expressions define regular languages. Closure under reversal; the powerset construction mapping NFAs to DFAs. Left and right languages; minimal DFAs. Brzozowski's minimization algorithm. Uniqueness up to isomorphism of minimal DFAs.

Chapter 1

Finite Automata using the Hereditarily Finite Sets

```
theory Finite_Automata_HF imports
  HereditarilyFinite.Ordinal
  "Regular-Sets.Regular_Exp"
begin
```

Finite Automata, both deterministic and non-deterministic, for regular languages. The Myhill-Nerode Theorem. Closure under intersection, concatenation, etc. Regular expressions define regular languages. Closure under reversal; the powerset construction mapping NFAs to DFAs. Left and right languages; minimal DFAs. Brzozowski's minimization algorithm. Uniqueness up to isomorphism of minimal DFAs.

1.1 Deterministic Finite Automata

Right invariance is the key property for equivalence relations on states of DFAs.

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

1.1.1 Basic Definitions

First, the record for DFAs

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

locale dfa =
  fixes M :: "'a dfa"
```

```
  assumes init [simp]: "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)"
begin

lemma finite_final [simp]: "finite (final M)"
  ⟨proof⟩
```

Transition function for a given starting state and word.

```
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}"
```

The left language WRT a state q is the set of words that lead to q.

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

Part of Prop 1 of Jean-Marc Champarnaud, A. Khorsi and T. Paranthoën, Split and join for minimizing: Brzozowski's algorithm, Prague Stringology Conference 2002

```
lemma left_lang_disjoint:
  "q1 ≠ q2 ⟹ left_lang q1 ∩ left_lang q2 = {}"
  ⟨proof⟩
```

The right language WRT a state q is the set of words that go from q to F.

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

lemma language_eq_right_lang: "language = right_lang (init M)"
  ⟨proof⟩

lemma nextl_app: "nextl q (xs@ys) = nextl (nextl q xs) ys"
  ⟨proof⟩

lemma nextl_snoc [simp]: "nextl q (xs@[x]) = nxt M (nextl q xs) x"
  ⟨proof⟩

lemma nextl_state: "q ∈ states M ⟹ nextl q xs ∈ states M"
  ⟨proof⟩

lemma nextl_init_state [simp]: "nextl (init M) xs ∈ states M"
  ⟨proof⟩
```

1.1.2 An Equivalence Relation on States

Two words are equivalent if they take the machine to the same state. See e.g. Kozen, Automata and Computability, Springer, 1997, page 90.

This relation asks, do u and v lead to the same state?

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

lemma equiv_eq_nextl: "equiv UNIV eq_nextl"
⟨*proof*⟩

lemma right_invariant_eq_nextl: "right_invariant eq_nextl"
⟨*proof*⟩

lemma range_nextl: "range (nextl (init M)) ⊆ states M"
⟨*proof*⟩

lemma eq_nextl_class_in_left_lang_im: "eq_nextl `` {u} ∈ left_lang ` states M"
⟨*proof*⟩

lemma language_eq_nextl: "language = eq_nextl `` (⋃q ∈ final M. left_lang q)"
⟨*proof*⟩

lemma finite_index_eq_nextl: "finite (UNIV // eq_nextl)"
⟨*proof*⟩

lemma index_eq_nextl_le_states: "card (UNIV // eq_nextl) ≤ card (states M)"
⟨*proof*⟩

1.1.3 Minimisation via Accessibility

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

lemma accessible_imp_states: "q ∈ accessible ⟹ q ∈ states M"
⟨*proof*⟩

lemma nxt_accessible: "q ∈ accessible ⟹ nxt M q a ∈ accessible"
⟨*proof*⟩

lemma inj_on_left_lang: "inj_on left_lang accessible"
⟨*proof*⟩

definition path_to :: "hf ⇒ 'a list" **where**
"path_to q ≡ SOME u. u ∈ left_lang q"

lemma path_to_left_lang: "q ∈ accessible ⟹ path_to q ∈ left_lang q"
⟨*proof*⟩

lemma nextl_path_to: "q ∈ accessible ⟹ nextl (dfa.init M) (path_to q) = q"
⟨*proof*⟩

definition Accessible_dfa :: "'a dfa" **where**
"Accessible_dfa = ⦇dfa.states = accessible,
init = init M,
final = final M ∩ accessible,
nxt = nxt M⦈"

lemma states_Accessible_dfa [simp]: "states Accessible_dfa = accessible"
⟨*proof*⟩

lemma init_Accessible_dfa [simp]: "init Accessible_dfa = init M"
⟨*proof*⟩

lemma final_Accessible_dfa [simp]: "final Accessible_dfa = final M ∩ accessible"
⟨*proof*⟩

lemma nxt_Accessible_dfa [simp]: "nxt Accessible_dfa = nxt M"
⟨*proof*⟩

interpretation Accessible: dfa Accessible_dfa
⟨*proof*⟩

lemma dfa_Accessible: "dfa Accessible_dfa"
⟨*proof*⟩

lemma nextl_Accessible_dfa [simp]:
"q ∈ accessible ⟹ Accessible.nextl q u = nextl q u"
⟨*proof*⟩

lemma init_Accessible: "init M ∈ accessible"
⟨*proof*⟩

declare nextl_Accessible_dfa [OF init_Accessible, simp]

lemma Accessible_left_lang_eq [simp]: "Accessible.left_lang q = left_lang q"
⟨*proof*⟩

lemma Accessible_right_lang_eq [simp]:
"q ∈ accessible ⟹ Accessible.right_lang q = right_lang q"
⟨*proof*⟩

lemma Accessible_language [simp]: "Accessible.language = language"
⟨*proof*⟩

lemma Accessible_accessible [simp]: "Accessible.accessible = accessible"
⟨*proof*⟩

lemma left_lang_half:
assumes sb: "⋃(left_lang ‘ qs1) ⊆ ⋃(left_lang ‘ qs2)"
and ne: "⋀x. x ∈ qs1 ⟹ left_lang x ≠ {}"
shows "qs1 ⊆ qs2"
⟨*proof*⟩

lemma left_lang_UN:
"⟦⋃(left_lang ‘ qs1) = ⋃(left_lang ‘ qs2); qs1 ∪ qs2 ⊆ accessible⟧
⟹ qs1 = qs2"
⟨*proof*⟩

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

1.1.4 An Equivalence Relation on States

Collapsing map on states. Two states are equivalent if they yield identical outcomes

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}"

lemma equiv_eq_right_lang: "equiv (states M) eq_right_lang"
⟨*proof*⟩

lemma eq_right_lang_finite_index: "finite (states M // eq_right_lang)"
⟨*proof*⟩

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})⦈"

lemma nxt_Collapse_resp: "(λq. eq_right_lang ‘‘ {nxt M q x}) respects
eq_right_lang"
⟨*proof*⟩

lemma finite_Collapse_state: "Q ∈ states M // eq_right_lang ⟹ finite

```
Q"
```
⟨*proof*⟩

interpretation `Collapse: dfa Collapse_dfa`
⟨*proof*⟩

corollary `dfa_Collapse: "dfa Collapse_dfa"`
⟨*proof*⟩

lemma `nextl_Collapse_dfa:`
```
    "Q = HF (eq_right_lang `` {q}) ⟹ Q ∈ dfa.states Collapse_dfa ⟹
     q ∈ states M ⟹
           Collapse.nextl Q u = HF (eq_right_lang `` {nextl q u})"
```
⟨*proof*⟩

lemma `ext_language_Collapse_dfa:`
```
    "u ∈ Collapse.language ⟷ u ∈ language"
```
⟨*proof*⟩

theorem `language_Collapse_dfa:`
```
    "Collapse.language = language"
```
⟨*proof*⟩

lemma `card_Collapse_dfa: "card (states M // eq_right_lang) ≤ card (states M)"`
⟨*proof*⟩

end

1.1.5 Isomorphisms Between DFAs

locale `dfa_isomorphism = M: dfa M + N: dfa N` **for** `M :: "'a dfa"` **and** `N :: "'a dfa" +`
```
  fixes h :: "hf ⇒ hf"
  assumes h: "bij_betw h (states M) (states N)"
      and init  [simp]: "h (init M) = init N"
      and final [simp]: "h ` final M = final N"
      and nxt   [simp]: "⋀q x. q ∈ states M ⟹ h (nxt M q x) = nxt N (h q) x"
```

begin

lemma `nextl [simp]: "q ∈ states M ⟹ h (M.nextl q u) = N.nextl (h q) u"`
⟨*proof*⟩

theorem `language: "M.language = N.language"`
⟨*proof*⟩

lemma nxt_inv_into: "q ∈ states N ⟹ nxt N q x = h (nxt M (inv_into (states M) h q) x)"
⟨*proof*⟩

lemma sym: "dfa_isomorphism N M (inv_into (states M) h)"
⟨*proof*⟩

lemma trans: "dfa_isomorphism N N' h' ⟹ dfa_isomorphism M N' (h' o h)"
⟨*proof*⟩

end

1.2 The Myhill-Nerode theorem: three characterisations of a regular language

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

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)"

This relation can be seen as an abstraction of the idea, do u and v lead to the same state? Compare with eq_nextl, which does precisely that.

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}"

lemma equiv_eq_app_right: "equiv UNIV (eq_app_right L)"
⟨*proof*⟩

lemma right_invariant_eq_app_right: "right_invariant (eq_app_right L)"
⟨*proof*⟩

lemma eq_app_right_eq: "eq_app_right L `` L = L"
⟨*proof*⟩

lemma MN_eq_app_right:
"finite (UNIV // eq_app_right L) ⟹ MyhillNerode L (eq_app_right L)"
⟨*proof*⟩

lemma MN_refines: "⟦MyhillNerode L R; (x,y) ∈ R⟧ ⟹ x ∈ L ⟷ y ∈ L"
⟨*proof*⟩

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

⟨*proof*⟩

Step 1 in the circle of implications: every regular language `L` is recognised by some Myhill-Nerode relation, `R`

context `dfa`
begin
lemma `MN_eq_nextl: "MyhillNerode language eq_nextl"`
⟨*proof*⟩

corollary `eq_nextl_refines_eq_app_right: "eq_nextl ⊆ eq_app_right language"`
⟨*proof*⟩

lemma `index_le_index_eq_nextl:`
`"card (UNIV // eq_app_right language) ≤ card (UNIV // eq_nextl)"`
⟨*proof*⟩

A specific lower bound on the number of states in a DFA

lemma `index_eq_app_right_lower:`
`"card (UNIV // eq_app_right language) ≤ card (states M)"`
⟨*proof*⟩
end

lemma `L1_2: "regular L ⟹ ∃R. MyhillNerode L R"`
⟨*proof*⟩

Step 2: every Myhill-Nerode relation `R` for the language `L` can be mapped to the canonical M-N relation.

lemma `L2_3:`
assumes `"MyhillNerode L R"`
obtains `"finite (UNIV // eq_app_right L)"`
`"card (UNIV // eq_app_right L) ≤ card (UNIV // R)"`
⟨*proof*⟩

Working towards step 3. Also, every Myhill-Nerode relation `R` for `L` can be mapped to a machine. The locale below constructs such a DFA.

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))"`
begin

lemma `injh: "inj_on h (UNIV//R)"`
⟨*proof*⟩

definition `hinv (<h⁻¹>)` **where** `"h⁻¹ ≡ inv_into (UNIV//R) h"`

lemma finix: "finite (UNIV//R)"
⟨*proof*⟩

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]})⦈"

lemma resp: "⋀x. (λu. R `` {u @ [x]}) respects R"
⟨*proof*⟩

lemma dfa: "dfa DFA"
⟨*proof*⟩

interpretation MN: dfa DFA
⟨*proof*⟩

lemma MyhillNerode: "MyhillNerode L R"
⟨*proof*⟩

lemma R_iff: "(∃x∈L. (u, x) ∈ R) = (u ∈ L)"
⟨*proof*⟩

lemma nextl: "MN.nextl (init DFA) u = h (R `` {u})"
⟨*proof*⟩

lemma language: "MN.language = L"
⟨*proof*⟩

lemma card_states: "card (states DFA) = card (UNIV // R)"
⟨*proof*⟩

end

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

corollary MN_imp_regular:
assumes "MyhillNerode L R" **shows** "regular L"
⟨*proof*⟩

corollary eq_app_right_finite_index_imp_dfa:
assumes "finite (UNIV // eq_app_right L)"

obtains M **where**
"dfa M" "dfa.language M = L" "card (states M) = card (UNIV // eq_app_right L)"
⟨*proof*⟩

Step 3

corollary L3_1: "finite (UNIV // eq_app_right L) ⟹ regular L"
⟨*proof*⟩

1.3 Non-Deterministic Finite Automata

These NFAs may include epsilon-transitions and multiple start states.

1.3.1 Basic Definitions

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

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)"
begin

lemma subset_states_finite [intro,simp]: "Q ⊆ states M ⟹ finite Q"
⟨*proof*⟩

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

lemma epsclo_eq_Image: "epsclo Q = states M ∩ (eps M)* `` Q"
⟨*proof*⟩

lemma epsclo_empty [simp]: "epsclo {} = {}"
⟨*proof*⟩

lemma epsclo_idem [simp]: "epsclo (epsclo Q) = epsclo Q"
⟨*proof*⟩

lemma epsclo_increasing: "Q ∩ states M ⊆ epsclo Q"
⟨*proof*⟩

lemma epsclo_Un [simp]: "epsclo (Q1 ∪ Q2) = epsclo Q1 ∪ epsclo Q2"

⟨*proof*⟩

lemma epsclo_UN [simp]: "epsclo (⋃x∈A. B x) = (⋃x∈A. epsclo (B x))"
⟨*proof*⟩

lemma epsclo_subset [simp]: "epsclo Q ⊆ states M"
⟨*proof*⟩

lemma epsclo_trivial [simp]: "eps M ⊆ Q × Q ⟹ epsclo Q = states M ∩ Q"
⟨*proof*⟩

lemma epsclo_mono: "Q' ⊆ Q ⟹ epsclo Q' ⊆ epsclo Q"
⟨*proof*⟩

lemma finite_epsclo [simp]: "finite (epsclo Q)"
⟨*proof*⟩

lemma finite_final: "finite (final M)"
⟨*proof*⟩

lemma finite_nxt: "q ∈ states M ⟹ finite (nxt M q x)"
⟨*proof*⟩

Transition function for a given starting state and word.

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 ≠ {}}"

The right language WRT a state q is the set of words that go from q to F.

definition right_lang :: "hf ⇒ 'a list set" **where**
"right_lang q ≡ {u. nextl {q} u ∩ final M ≠ {}}"

lemma nextl_epsclo [simp]: "nextl (epsclo Q) xs = nextl Q xs"
⟨*proof*⟩

lemma epsclo_nextl [simp]: "epsclo (nextl Q xs) = nextl Q xs"
⟨*proof*⟩

lemma nextl_app: "nextl Q (xs@ys) = nextl (nextl Q xs) ys"
⟨*proof*⟩

lemma nextl_snoc [simp]: "nextl Q (xs@[x]) = (⋃q ∈ nextl Q xs. epsclo (nxt M q x))"
⟨*proof*⟩

lemma nextl_state: "nextl Q xs ⊆ states M"
⟨*proof*⟩

lemma nextl_mono: "Q' ⊆ Q ⟹ nextl Q' u ⊆ nextl Q u"
⟨*proof*⟩

lemma nextl_eps: "q ∈ nextl Q u ⟹ (q,q') ∈ eps M ⟹ q' ∈ states M
⟹ q' ∈ nextl Q u"
⟨*proof*⟩

lemma finite_nextl: "finite (nextl Q u)"
⟨*proof*⟩

lemma nextl_empty [simp]: "nextl {} xs = {}"
⟨*proof*⟩

lemma nextl_Un: "nextl (Q1 ∪ Q2) xs = nextl Q1 xs ∪ nextl Q2 xs"
⟨*proof*⟩

lemma nextl_UN: "nextl (⋃i∈I. f i) xs = (⋃i∈I. nextl (f i) xs)"
⟨*proof*⟩

1.3.2 The Powerset Construction

definition Power_dfa :: "'a dfa" **where**
"Power_dfa = ⦇dfa.states = {HF (epsclo q) | q. q ∈ 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))⦈"

lemma states_Power_dfa [simp]: "dfa.states Power_dfa = HF ‘ epsclo ‘ Pow (states M)"
⟨*proof*⟩

lemma init_Power_dfa [simp]: "dfa.init Power_dfa = HF (epsclo (nfa.init M))"
⟨*proof*⟩

lemma final_Power_dfa [simp]: "dfa.final Power_dfa = {HF (epsclo Q) | Q. Q ⊆ states M ∧ Q ∩ final M ≠ {}}"
⟨*proof*⟩

lemma nxt_Power_dfa [simp]: "dfa.nxt Power_dfa = (λQ x. HF(⋃q ∈ epsclo (hfset Q). epsclo (nxt M q x)))"
⟨*proof*⟩

interpretation Power: dfa Power_dfa

⟨*proof*⟩

corollary dfa_Power: "dfa Power_dfa"
⟨*proof*⟩

lemma nextl_Power_dfa:
"qs ∈ dfa.states Power_dfa
⟹ dfa.nextl Power_dfa qs u = HF (⋃q ∈ hfset qs. nextl {q} u)"
⟨*proof*⟩

Part of Prop 4 of Jean-Marc Champarnaud, A. Khorsi and T. Paranthoën (2002)

lemma Power_right_lang:
"qs ∈ dfa.states Power_dfa ⟹ Power.right_lang qs = (⋃q ∈ hfset qs. right_lang q)"
⟨*proof*⟩

The Power DFA accepts the same language as the NFA.

theorem Power_language [simp]: "Power.language = language"
⟨*proof*⟩

Every language accepted by a NFA is also accepted by a DFA.

corollary imp_regular: "regular language"
⟨*proof*⟩

end

As above, outside the locale

corollary nfa_imp_regular:
assumes "nfa M" "nfa.language M = L"
shows "regular L"
⟨*proof*⟩

1.4 Closure Properties for Regular Languages

1.4.1 The Empty Language

theorem regular_empty: "regular {}"
⟨*proof*⟩

1.4.2 The Empty Word

theorem regular_nullstr: "regular {[]}"
⟨*proof*⟩

1.4.3 Single Symbol Languages

theorem regular_singstr: "regular {[a]}"
⟨*proof*⟩

1.4.4 The Complement of a Language

theorem regular_Compl:
  assumes S: "regular S" **shows** "regular (-S)"
⟨*proof*⟩

1.4.5 The Intersection and Union of Two Languages

By the familiar product construction

theorem regular_Int:
  assumes S: "regular S" **and** T: "regular T" **shows** "regular (S ∩ T)"
⟨*proof*⟩

corollary regular_Un:
  assumes S: "regular S" **and** T: "regular T" **shows** "regular (S ∪ T)"
  ⟨*proof*⟩

1.4.6 The Concatenation of Two Languages

lemma Inlr_rtrancl [simp]: "((λq. (Inl q, Inr a)) ‘ A)* = ((λq. (Inl q, Inr a)) ‘ A)="
  ⟨*proof*⟩

theorem regular_conc:
  assumes S: "regular S" **and** T: "regular T" **shows** "regular (S @@ T)"
⟨*proof*⟩

lemma regular_word: "regular {u}"
⟨*proof*⟩

All finite sets are regular.

theorem regular_finite: "finite L ⟹ regular L"
  ⟨*proof*⟩

1.4.7 The Kleene Star of a Language

theorem regular_star:
  assumes S: "regular S" **shows** "regular (star S)"
⟨*proof*⟩

1.4.8 The Reversal of a Regular Language

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. {q' ∈ dfa.states MS. q = dfa.nxt MS q' x},
                      eps   = {}|)"

lemma states_Reverse_nfa [simp]: "states (Reverse_nfa MS) = dfa.states MS"
⟨*proof*⟩

lemma init_Reverse_nfa [simp]: "init (Reverse_nfa MS) = dfa.final MS"
⟨*proof*⟩

lemma final_Reverse_nfa [simp]: "final (Reverse_nfa MS) = {dfa.init MS}"
⟨*proof*⟩

lemma nxt_Reverse_nfa [simp]:
"nxt (Reverse_nfa MS) q x = {q' ∈ dfa.states MS. q = dfa.nxt MS q' x}"
⟨*proof*⟩

lemma eps_Reverse_nfa [simp]: "eps (Reverse_nfa MS) = {}"
⟨*proof*⟩

context dfa
begin

lemma nfa_Reverse_nfa: "nfa (Reverse_nfa M)"
⟨*proof*⟩

lemma nextl_Reverse_nfa:
"nfa.nextl (Reverse_nfa M) Q u = {q' ∈ dfa.states M. dfa.nextl M q' (rev u) ∈ Q}"
⟨*proof*⟩

Part of Prop 3 of Jean-Marc Champarnaud, A. Khorsi and T. Paranthoën (2002)

lemma right_lang_Reverse: "nfa.right_lang (Reverse_nfa M) q = rev ‘ (dfa.left_lang M q)"
⟨*proof*⟩

lemma right_lang_Reverse_disjoint:
"q1 ≠ q2 ⟹ nfa.right_lang (Reverse_nfa M) q1 ∩ nfa.right_lang (Reverse_nfa M) q2 = {}"
⟨*proof*⟩

lemma epsclo_Reverse_nfa [simp]: "nfa.epsclo (Reverse_nfa M) Q = Q ∩ dfa.states M"
⟨*proof*⟩

theorem language_Reverse_nfa [simp]:
"nfa.language (Reverse_nfa M) = (rev ‘ dfa.language M)"
⟨*proof*⟩

end

corollary regular_Reverse:
  assumes S: "regular S" **shows** "regular (rev ‘ S)"
⟨*proof*⟩

All regular expressions yield regular languages.

theorem regular_lang: "regular (lang r)"
  ⟨*proof*⟩

1.5 Brzozowski's Minimization Algorithm

context dfa
  begin

1.5.1 More about the relation eq_app_right

  lemma left_eq_app_right:
      "⟦u ∈ left_lang q; v ∈ left_lang q⟧ ⟹ (u,v) ∈ eq_app_right language"
    ⟨*proof*⟩

  lemma eq_app_right_class_eq:
    "UNIV // eq_app_right language = (λq. eq_app_right language ‘‘ {path_to q}) ‘ accessible"
  ⟨*proof*⟩

  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)"
    ⟨*proof*⟩

  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"
    ⟨*proof*⟩

A minimal machine has a minimal number of states, compared with any other machine for the same language.

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

  definition index_f :: "'a list set ⇒ hf" **where**
    "index_f ≡ SOME h. bij_betw h (UNIV // eq_app_right language) (hfset (ord_of min_states))"

lemma index_f: "bij_betw index_f (UNIV // eq_app_right language) (hfset (ord_of min_states))"
⟨*proof*⟩

interpretation Canon:
MyhillNerode_dfa language "eq_app_right language"
language
min_states index_f
⟨*proof*⟩

interpretation MN: dfa Canon.DFA
⟨*proof*⟩

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

theorem minimal_imp_isomorphic_to_canonical:
assumes minimal
shows "dfa_isomorphism M Canon.DFA iso"
⟨*proof*⟩

lemma states_PR [simp]:
"dfa.states (nfa.Power_dfa (Reverse_nfa M)) = HF ` Pow (dfa.states M)"
⟨*proof*⟩

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)))"
⟨*proof*⟩

abbreviation APR :: "'x dfa ⇒ 'x dfa" **where**
"APR X ≡ dfa.Accessible_dfa (nfa.Power_dfa (Reverse_nfa X))"

theorem minimal_APR:
assumes "dfa.states M = accessible"
shows "dfa.minimal (APR M)"
⟨*proof*⟩

definition Brzozowski :: "'a dfa" **where**
"Brzozowski ≡ APR (APR M)"

lemma dfa_Brzozowski: "dfa Brzozowski"
⟨*proof*⟩

theorem language_Brzozowski: "dfa.language Brzozowski = language"
⟨*proof*⟩

theorem minimal_Brzozowski: "dfa.minimal Brzozowski"
⟨*proof*⟩

end

lemma index_f_cong:
"⟦dfa.language M = dfa.language N; dfa M; dfa N⟧ ⟹ dfa.index_f M = dfa.index_f N"
⟨*proof*⟩

theorem minimal_imp_isomorphic:
"⟦dfa.language M = dfa.language N; dfa.minimal M; dfa.minimal N; dfa M; dfa N⟧
⟹ ∃h. dfa_isomorphism M N h"
⟨*proof*⟩

end