

# 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.

Initially this formalization was based on automata whose state type was always `hf`. In a revision, it was generalized: many of the constructions are now based on automata with a polymorphic state type `'s`.

### 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

We try to make as much as possible polymorphic in the state space `'s` and independent of type `hf` to increase reusability.

First, the record for DFAs

```

record ('a,'s) dfa = states :: "'s set"
    init    :: "'s"
    final   :: "'s set"
    nxt     :: "'s ⇒ 'a ⇒ 's"

locale dfa =
  fixes M :: "('a,'s) 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)"
  using final finite_subset finite by blast

Transition function for a given starting state and word.
primrec nextl :: "[ 's, 'a list ] ⇒ 's" 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 :: "'s ⇒ '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 = {}"
  unfolding left_lang_def by auto

The right language WRT a state q is the set of words that go from q to F.
definition right_lang :: "'s ⇒ 'a list set" where
  "right_lang q ≡ {u. nextl q u ∈ final M}"

lemma language_eq_right_lang: "language = right_lang (init M)"
  using language_def right_lang_def by auto

lemma nextl_app: "nextl q (xs@ys) = nextl (nextl q xs) ys"
  by (induct xs arbitrary: q) auto

lemma nextl_snoc [simp]: "nextl q (xs@[x]) = nxt M (nextl q xs) x"
  by (simp add: nextl_app)

```

```
lemma nextl_state: "q ∈ states M ⇒ nextl q xs ∈ states M"
  by (induct xs arbitrary: q) (auto simp: nxt)
```

```
lemma nextl_init_state [simp]: "nextl (init M) xs ∈ states M"
  by (simp add: nextl_state)
```

### 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"
  by (simp add: equivI refl_on_def sym_def trans_def eq_nextl_def)
```

```
lemma right_invariant_eq_nextl: "right_invariant eq_nextl"
  by (auto simp: right_invariant_def eq_nextl_def nextl_app)
```

```
lemma range_nextl: "range (nextl (init M)) ⊆ states M"
  using hmem_def nextl_init_state by auto
```

```
lemma eq_nextl_class_in_left_lang_im: "eq_nextl ‘‘ {u} ∈ left_lang ‘
states M"
  apply (rule rev_image_eqI [of "nextl (init M) u"])
  apply (auto simp: eq_nextl_def left_lang_def)
  done
```

```
lemma language_eq_nextl: "language = eq_nextl ‘‘ (⋃ q ∈ final M. left_lang
q)"
  by (auto simp: language_def eq_nextl_def left_lang_def hmem_def)
```

```
lemma finite_index_eq_nextl: "finite (UNIV // eq_nextl)"
  apply (rule finite_surj [where f = left_lang, OF finite])
  apply (auto simp: quotient_def eq_nextl_class_in_left_lang_im)
  done
```

```
lemma index_eq_nextl_le_states: "card (UNIV // eq_nextl) ≤ card (states
M)"
  apply (rule surj_card_le [where f = left_lang, OF finite])
  apply (auto simp: quotient_def eq_nextl_class_in_left_lang_im)
  done
```

### 1.1.3 Minimisation via Accessibility

```
definition accessible :: "'s set" where
  "accessible ≡ {q. left_lang q ≠ {}}"
```

```

lemma accessible_imp_states: "q ∈ accessible ⇒ q ∈ states M"
  by (auto simp: accessible_def left_lang_def)

lemma nxt_accessible: "q ∈ accessible ⇒ nxt M q a ∈ accessible"
  by (auto simp: image_iff accessible_def left_lang_def) (metis nextl.simps
nextl_app)

lemma inj_on_left_lang: "inj_on left_lang accessible"
  by (auto simp: inj_on_def left_lang_def accessible_def)

definition path_to :: "'s ⇒ '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"
  unfolding path_to_def left_lang_def accessible_def
  by (auto intro: someI)

lemma nextl_path_to: "q ∈ accessible ⇒ nextl (dfa.init M) (path_to
q) = q"
  using path_to_left_lang unfolding left_lang_def
  by auto

definition Accessible_dfa :: "('a,'s) 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"
  by (simp add: Accessible_dfa_def)

lemma init_Accessible_dfa [simp]: "init Accessible_dfa = init M"
  by (simp add: Accessible_dfa_def)

lemma final_Accessible_dfa [simp]: "final Accessible_dfa = final M ∩
accessible"
  by (simp add: Accessible_dfa_def)

lemma nxt_Accessible_dfa [simp]: "nxt Accessible_dfa = nxt M"
  by (simp add: Accessible_dfa_def)

interpretation Accessible: dfa Accessible_dfa
proof unfold_locales
  show "init Accessible_dfa ∈ states Accessible_dfa"
    by (auto simp: accessible_def left_lang_def) (metis nextl.simps(1))
next
  show "final Accessible_dfa ⊆ states Accessible_dfa"
    by (auto simp: accessible_imp_states)

```

```

next
  fix q a
  show "q ∈ states Accessible_dfa ⇒ nxt Accessible_dfa q a ∈ states
Accessible_dfa"
  by (auto simp: nxt nxt_accessible)
next
  show "finite (states Accessible_dfa)"
  by (auto intro: accessible_imp_states finite_subset finite)
qed

lemma dfa_Accessible: "dfa Accessible_dfa"
  by unfold_locales

lemma nextl_Accessible_dfa [simp]:
  "q ∈ accessible ⇒ Accessible.nextl q u = nextl q u"
  by (induct u rule: List.rev_induct, auto)

lemma init_Accessible: "init M ∈ accessible"
  using dfa.init dfa_Accessible by force

declare nextl_Accessible_dfa [OF init_Accessible, simp]

lemma Accessible_left_lang_eq [simp]: "Accessible.left_lang q = left_lang
q"
  by (auto simp: Accessible.left_lang_def left_lang_def)

lemma Accessible_right_lang_eq [simp]:
  "q ∈ accessible ⇒ Accessible.right_lang q = right_lang q"
  by (auto simp: Accessible.right_lang_def right_lang_def accessible_def
left_lang_def nextl_app [symmetric])

lemma Accessible_language [simp]: "Accessible.language = language"
  by (simp add: Accessible.language_eq_right_lang language_eq_right_lang
Accessible_right_lang_eq [OF init_Accessible])

lemma Accessible_accessible [simp]: "Accessible.accessible = accessible"
  using Accessible.accessible_def accessible_def by auto

lemma left_lang_half:
  assumes sb: " $\bigcup (\text{left\_lang } ' \text{ qs1}) \subseteq \bigcup (\text{left\_lang } ' \text{ qs2})$ "
  and ne: " $\bigwedge x. x \in \text{qs1} \Rightarrow \text{left\_lang } x \neq \{\}$ "
  shows " $\text{qs1} \subseteq \text{qs2}$ "
proof
  fix x
  assume x: "x ∈ qs1"
  with ne obtain y where y: "y ∈ left_lang x"
  by blast
  then have "y ∈  $\bigcup (\text{left\_lang } ' \text{ qs2})$ "

```

```

    using x sb [THEN subsetD]
  by blast
  then show "x ∈ qs2"
  by (metis UN_E disjoint_iff_not_equal left_lang_disjoint y)
qed

lemma left_lang_UN:
  "[⋃ (left_lang ' qs1) = ⋃ (left_lang ' qs2); qs1 ∪ qs2 ⊆ accessible]
  ⇒ qs1 = qs2"
  apply (rule equalityI [OF _ dfa.left_lang_half [OF dfa_Accessible]])
  apply (rule dfa.left_lang_half [OF dfa_Accessible], auto simp: accessible_def)
  done

```

```

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 :: "('s × 's) 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"
  by (auto simp: equiv_def refl_on_def sym_def trans_def eq_right_lang_def)

```

```

lemma eq_right_lang_finite_index: "finite (states M // eq_right_lang)"
  by (metis finite_imageI finite_proj_image)

```

end

Now we need to specialize to hf states.

```

type_synonym 'a dfa_hf = "('a,hf) dfa"

```

```

locale dfa_hf = dfa M for M :: "'a dfa_hf"
begin

```

```

definition Collapse_dfa :: "'a dfa_hf" 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"

```

```

    by (auto simp: nextl.simps [symmetric] congruent_def eq_right_lang_def
next right_lang_def
      simp del: nextl.simps)

lemma finite_Collapse_state: "Q ∈ states M // eq_right_lang ⇒ finite
Q"
  by (meson equiv_eq_right_lang finite_subset in_quotient_imp_subset finite)

interpretation Collapse: dfa Collapse_dfa
proof unfold_locales
  show "dfa.init Collapse_dfa ∈ dfa.states Collapse_dfa"
    by (simp add: Collapse_dfa_def quotientI)
next
  show "dfa.final Collapse_dfa ⊆ dfa.states Collapse_dfa"
    using final
    by (auto simp: Collapse_dfa_def quotientI)
next
  fix q a
  show "q ∈ dfa.states Collapse_dfa ⇒ dfa.next Collapse_dfa q a ∈ dfa.states
Collapse_dfa"
    by (auto simp: Collapse_dfa_def next quotientI finite_Collapse_state
      UN_equiv_class_type [OF equiv_eq_right_lang next_Collapse_resp])
next
  show "finite (dfa.states Collapse_dfa)"
    by (simp add: Collapse_dfa_def eq_right_lang_finite_index)
qed

corollary dfa_Collapse: "dfa Collapse_dfa"
  by unfold_locales

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})"
  apply (induct u rule: List.rev_induct, auto)
  apply (simp add: Collapse_dfa_def)
  apply (subst inj_on_eq_iff [OF inj_on_HF])
  apply (rule UN_equiv_class_type [OF equiv_eq_right_lang next_Collapse_resp])
  apply (auto simp: next quotientI finite_Collapse_state nextl_state)
  apply (force simp add: nextl.simps [symmetric] eq_right_lang_def next
right_lang_def
      simp del: nextl.simps)
  apply (metis equiv_class_self equiv_eq_right_lang nextl_state)
  done

lemma ext_language_Collapse_dfa:
  "u ∈ Collapse.language ⇔ u ∈ language"
  apply (simp add: Collapse.language_def language_def)
  apply (subst nextl_Collapse_dfa)

```

```

apply (auto simp: Collapse_dfa_def)
using final [THEN subsetD] init
apply (auto simp: quotientI inj_on_eq_iff [OF inj_on_HF] finite_Collapse_state
        UN_equiv_class_type [OF equiv_eq_right_lang nxt_Collapse_resp])
apply (drule eq_equiv_class [OF _ equiv_eq_right_lang])
apply (auto simp: eq_right_lang_def right_lang_def)
apply (metis mem_Collect_eq nextl.simps(1))
done

theorem language_Collapse_dfa:
  "Collapse.language = language"
  by (simp add: ext_language_Collapse_dfa subset_antisym subset_iff)

lemma card_Collapse_dfa: "card (states M // eq_right_lang) ≤ card (states
M)"
  by (metis card_image_le finite proj_image)

end

```

### 1.1.5 Isomorphisms Between DFAs

```

locale dfa_isomorphism = M: dfa M + N: dfa N for M :: "('a,'sm) dfa" and
N :: "('a,'sn) dfa" +
  fixes h :: "'sm ⇒ 'sn"
  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"
  by (induct u rule: List.rev_induct) (auto simp: M.nextl_state)

theorem language: "M.language = N.language"
proof (rule set_eqI)
  fix u
  have "M.nextl (init M) u ∈ final M ↔ h (M.nextl (init M) u) ∈ h `
final M"
    by (subst inj_on_image_mem_iff [OF bij_betw_imp_inj_on [OF h] _ M.final])
(auto)
  also have "... ↔ N.nextl (init N) u ∈ final N"
    by simp
  finally show "u ∈ M.language ↔ u ∈ dfa.language N"
    by (simp add: M.language_def N.language_def)
qed

```

```

lemma nxt_inv_into: "q ∈ states N  $\implies$  nxt N q x = h (nxt M (inv_into
(states M) h q) x)"
  apply (subst nxt)
  apply (metis bij_betw_def h inv_into_into)
  apply (metis bij_betw_inv_into_right h)
  done

```

```

lemma sym: "dfa_isomorphism N M (inv_into (states M) h)"
  apply unfold_locales
  apply (metis bij_betw_inv_into h)
  apply (metis M.init bij_betw_imp_inj_on h inv_into_f_eq init)
  apply (metis M.final bij_betw_def bij_betw_inv_into_subset h final)
  apply (simp add: nxt_inv_into)
  apply (metis M.nxt bij_betw_def h inv_into_f_eq inv_into_into)
  done

```

```

lemma trans: "dfa_isomorphism N N' h'  $\implies$  dfa_isomorphism M N' (h' o
h)"
  apply (auto simp: dfa_isomorphism_def dfa_isomorphism_axioms_def)
  apply unfold_locales
  apply (metis bij_betw_comp_iff h)
  apply (metis imageI final)
  apply (simp only: final [symmetric] image_comp) apply simp
  apply (metis bij_betw_def h image_iff)
  done

```

end

In order to transition between 's and hf, we use bijections:

```

lemma inj_on_finite_hf:
  <finite S  $\implies$   $\exists$ f:: 's  $\Rightarrow$  hf. inj_on f S>
by (meson comp_inj_on finite_imp_inj_to_nat_seg inj_ord_of)

```

```

lemma bij_betw_finite_hf:
  <finite S  $\implies$   $\exists$ f:: 's  $\Rightarrow$  hf. bij_betw f S (f ' S)>
by (meson inj_on_finite_hf inj_on_imp_bij_betw)

```

There is always an isomorphism between a ('a, 's) dfa and a 'a dfa\_hf:

```

context dfa
begin

```

```

definition bij_s_hf :: "'s  $\Rightarrow$  hf" where
"bij_s_hf = (SOME f :: 's  $\Rightarrow$  hf. bij_betw f (dfa.states M) (f ' dfa.states
M))"

```

```

lemma bij_betw_bij_s_hf: "bij_betw bij_s_hf (dfa.states M) (bij_s_hf
' dfa.states M)"
unfolding bij_s_hf_def using bij_betw_finite_hf[OF finite]
by (metis (no_types, lifting) someI_ex)

```

```

abbreviation bij_s_hf_M :: "'a dfa_hf" where
  "bij_s_hf_M  $\equiv$  ( $\lfloor$  dfa.states = bij_s_hf ' dfa.states M,
    dfa.init = bij_s_hf (dfa.init M),
    dfa.final = bij_s_hf ' dfa.final M,
    dfa.nxt = ( $\lambda$ q x. bij_s_hf (dfa.nxt M (the_inv_into (dfa.states
M) bij_s_hf q) x))  $\rfloor$ "

```

```

lemma dfa_bij_s_hf_M: "dfa bij_s_hf_M"

```

```

proof

```

```

  fix q x

```

```

  assume "q  $\in$  states bij_s_hf_M"

```

```

  then show "nxt bij_s_hf_M q x  $\in$  states bij_s_hf_M"

```

```

    by (metis bij_betw_apply bij_betw_bij_s_hf bij_betw_the_inv_into nxt
      select_convs(1,4))

```

```

qed (use finite final in auto)

```

```

interpretation M_iso: dfa_isomorphism M bij_s_hf_M bij_s_hf

```

```

proof (intro dfa_isomorphism.intro dfa_axioms dfa_bij_s_hf_M dfa_isomorphism_axioms.intro)

```

```

  fix q x

```

```

  assume "q  $\in$  states M"

```

```

  then show "bij_s_hf (nxt M q x) = nxt bij_s_hf_M (bij_s_hf q) x"

```

```

    by (metis bij_betw_bij_s_hf bij_betw_def select_convs(4) the_inv_into_f_f)

```

```

qed (use bij_betw_bij_s_hf in auto)

```

```

lemmas L_M_eq_L_bij_s_hf_M = M_iso.language

```

```

end

```

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

```

definition regular :: "'a list set  $\Rightarrow$  bool" where

```

```

  "regular L  $\equiv$   $\exists$ M :: 'a dfa_hf. dfa M  $\wedge$  dfa.language M = L"

```

```

definition MyhillNerode :: "'a list set  $\Rightarrow$  ('a list * 'a list) set  $\Rightarrow$  bool"
where

```

```

  "MyhillNerode L R  $\equiv$  equiv UNIV R  $\wedge$  right_invariant R  $\wedge$  finite (UNIV//R)
 $\wedge$  ( $\exists$ 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_next1`, which does precisely that.

```

definition eq_app_right :: "'a list set  $\Rightarrow$  ('a list * 'a list) set" where

```

```

  "eq_app_right L  $\equiv$  {(u,v).  $\forall$ w. u@w  $\in$  L  $\longleftrightarrow$  v@w  $\in$  L}"

```

```

lemma equiv_eq_app_right: "equiv UNIV (eq_app_right L)"

```

```

  by (simp add: equivI refl_on_def sym_def trans_def eq_app_right_def)

```

```
lemma right_invariant_eq_app_right: "right_invariant (eq_app_right L)"
  by (simp add: right_invariant_def eq_app_right_def)
```

```
lemma eq_app_right_eq: "eq_app_right L `` L = L"
  unfolding eq_app_right_def
  by auto (metis append_Nil2)
```

```
lemma MN_eq_app_right:
  "finite (UNIV // eq_app_right L)  $\implies$  MyhillNerode L (eq_app_right L)"
  unfolding MyhillNerode_def
  using eq_app_right_eq
  by (auto simp: equiv_eq_app_right right_invariant_eq_app_right)
```

```
lemma MN_refines: "[MyhillNerode L R; (x,y)  $\in$  R]  $\implies$  x  $\in$  L  $\longleftrightarrow$  y  $\in$  L"
  unfolding equiv_def trans_def sym_def MyhillNerode_def
  by blast
```

```
lemma MN_refines_eq_app_right: "MyhillNerode L R  $\implies$  R  $\subseteq$  eq_app_right L"
  unfolding eq_app_right_def MyhillNerode_def right_invariant_def equiv_def
  trans_def sym_def
  by blast
```

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

```
context dfa
begin
```

```
lemma regular_dfa: "regular language"
  unfolding regular_def L_M_eq_L_bij_s_hf_M using dfa_bij_s_hf_M
  by blast
```

```
lemma MN_eq_nextl: "MyhillNerode language eq_nextl"
  unfolding MyhillNerode_def
  using language_eq_nextl
  by (blast intro: equiv_eq_nextl right_invariant_eq_nextl finite_index_eq_nextl)
```

```
corollary eq_nextl_refines_eq_app_right: "eq_nextl  $\subseteq$  eq_app_right language"
  by (simp add: MN_eq_nextl MN_refines_eq_app_right)
```

```
lemma index_le_index_eq_nextl:
  "card (UNIV // eq_app_right language)  $\leq$  card (UNIV // eq_nextl)"
  by (metis finite_refines_card_le finite_index_eq_nextl equiv_eq_nextl
  equiv_eq_app_right
  eq_nextl_refines_eq_app_right)
```

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)"
    using index_eq_nextl_le_states index_le_index_eq_nextl order_trans
  by blast
end

```

```

lemma L1_2: "regular L  $\implies$   $\exists$ R. MyhillNerode L R"
  by (metis dfa.MN_eq_nextl regular_def)

```

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)"
  by (meson assms MN_refines_eq_app_right MyhillNerode_def equiv_eq_app_right
    finite_refines_finite finite_refines_card_le)

```

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  $\Rightarrow$  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)"
    using h bij_betw_imp_inj_on by blast

  definition hinv (<math>h^{-1}>) where "h^{-1}  $\equiv$  inv_into (UNIV//R) h"

  lemma finix: "finite (UNIV//R)"
    using h bij_betw_finite finite_hfset by blast

  definition DFA :: "'a dfa_hf" where
    "DFA = (states = h ' (UNIV//R),
      init = h (R ' {[]}),
      final = {h (R ' {u}) | u. u  $\in$  A},
      nxt =  $\lambda$ q x. h ( $\bigcup$ u  $\in$  h^{-1} q. R ' {u@[x]}))"

  lemma resp: " $\wedge$ x. ( $\lambda$ u. R ' {u @ [x]}) respects R"
    using riR
    by (auto simp: congruent_def right_invariant_def intro!: equiv_class_eq
      [OF eqR])

  lemma dfa: "dfa DFA"
    apply (auto simp: dfa_def DFA_def quotientI finix)

```

```

    apply (subst inj_on_image_mem_iff [OF injh])
    apply (rule UN_equiv_class_type [OF eqR resp])
    apply (auto simp: injh DFA_def hinv_def quotientI)
  done

interpretation MN: dfa DFA
  by (simp add: dfa)

lemma MyhillNerode: "MyhillNerode L R"
  using L
  by (auto simp: MyhillNerode_def eqR riR finix)

lemma R_iff: "( $\exists x \in L. (u, x) \in R$ ) = (u  $\in$  L)"
  using MN_refines MyhillNerode eqR eq_equiv_class_iff
  by fastforce

lemma next1: "MN.next1 (init DFA) u = h (R ‘ {u})"
  apply (induct u rule: List.rev_induct, auto)
  apply (simp_all add: DFA_def hinv_def injh quotientI UN_equiv_class
[OF eqR resp])
  done

lemma language: "MN.language = L"
proof -
  { fix u
    have "u  $\in$  MN.language  $\longleftrightarrow$  u  $\in$  L"
      using L eqR
      apply (simp add: MN.language_def next1)
      apply (simp add: DFA_def inj_on_eq_iff [OF injh] eq_equiv_class_iff
[OF eqR] quotientI)
      apply (auto simp: equiv_def sym_def)
    done
  } then show ?thesis
    by blast
qed

lemma card_states: "card (states DFA) = card (UNIV // R)"
  using h
  by (simp add: DFA_def bij_betw_def) (metis card_image)

end

theorem MN_imp_dfa:
  assumes "MyhillNerode L R"
  obtains M where "dfa_hf M" "dfa.language M = L" "card (states M) =
card (UNIV//R)"
proof -
  from assms obtain A

```

```

    where eqR: "equiv UNIV R"
      and riR: "right_invariant R"
      and finix: "finite (UNIV//R)"
      and L: "L = R 'A"
    by (auto simp: MyhillNerode_def)
let ?n = "card (UNIV//R)"
from ex_bij_betw_finite_nat [OF finix]
obtain h where h: "bij_betw h (UNIV//R) (hfset (ord_of ?n))"
  using bij_betw_ord_ofI by blast
interpret MN: MyhillNerode_dfa L R A "?n" h
  by (simp add: MyhillNerode_dfa_def eqR riR L h)
show ?thesis
  using MN.card_states MN.dfa MN.language dfa_hf.intro that by blast
qed

```

```

corollary MN_imp_regular:
  assumes "MyhillNerode L R" shows "regular L"
  unfolding regular_def by (metis dfa_hf_def MN_imp_dfa[OF assms])

```

```

corollary eq_app_right_finite_index_imp_dfa:
  assumes "finite (UNIV // eq_app_right L)"
  obtains M where
    "dfa_hf M" "dfa.language M = L" "card (states M) = card (UNIV // eq_app_right
L)"
  using MN_eq_app_right MN_imp_dfa assms by blast

```

Step 3

```

corollary L3_1: "finite (UNIV // eq_app_right L)  $\implies$  regular L"
  unfolding regular_def by (metis eq_app_right_finite_index_imp_dfa dfa_hf_def)

```

## 1.3 Non-Deterministic Finite Automata

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

### 1.3.1 Basic Definitions

```

record ('a,'s) nfa = states :: "'s set"
  init      :: "'s set"
  final     :: "'s set"
  nxt       :: "'s  $\implies$  'a  $\implies$  's set"
  eps       :: "('s * 's) set"

locale nfa =
  fixes M :: "('a,'s) nfa"
  assumes init: "init M  $\subseteq$  states M"
    and final: "final M  $\subseteq$  states M"
    and nxt: " $\bigwedge q x. q \in \text{states } M \implies \text{nxt } M \ q \ x \subseteq \text{states } M$ "
    and finite: "finite (states M)"

```

```

begin

lemma subset_states_finite [intro,simp]: "Q ⊆ states M ⇒ finite Q"
  by (simp add: finite_subset finite)

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

lemma epsclo_eq_Image: "epsclo Q = states M ∩ (eps M)* `` Q"
  by (auto simp: epsclo_def)

lemma epsclo_empty [simp]: "epsclo {} = {}"
  by (auto simp: epsclo_def)

lemma epsclo_idem [simp]: "epsclo (epsclo Q) = epsclo Q"
  by (auto simp: epsclo_def)

lemma epsclo_increasing: "Q ∩ states M ⊆ epsclo Q"
  by (auto simp: epsclo_def)

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

lemma epsclo_UN [simp]: "epsclo (⋃ x∈A. B x) = (⋃ x∈A. epsclo (B x))"
  by (auto simp: epsclo_def)

lemma epsclo_subset [simp]: "epsclo Q ⊆ states M"
  by (auto simp: epsclo_def)

lemma epsclo_trivial [simp]: "eps M ⊆ Q × Q ⇒ epsclo Q = states M
∩ Q"
  by (auto simp: epsclo_def elim: rtranclE)

lemma epsclo_mono: "Q' ⊆ Q ⇒ epsclo Q' ⊆ epsclo Q"
  by (auto simp: epsclo_def)

lemma finite_epsclo [simp]: "finite (epsclo Q)"
  using epsclo_subset finite_subset finite by blast

lemma finite_final: "finite (final M)"
  using final finite_subset finite by blast

lemma finite_nxt: "q ∈ states M ⇒ finite (nxt M q x)"
  by (metis finite_subset finite nxt)

Transition function for a given starting state and word.

primrec nextl :: "[ 's set, 'a list ] ⇒ 's 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  $\equiv$  {xs. nextl (init M) xs  $\cap$  final M  $\neq$  {}}"

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

definition right_lang :: "'s  $\Rightarrow$  'a list set" where
  "right_lang q  $\equiv$  {u. nextl {q} u  $\cap$  final M  $\neq$  {}}"

lemma nextl_epsclo [simp]: "nextl (epsclo Q) xs = nextl Q xs"
  by (induct xs) auto

lemma epsclo_nextl [simp]: "epsclo (nextl Q xs) = nextl Q xs"
  by (induct xs arbitrary: Q) auto

lemma nextl_app: "nextl Q (xs@ys) = nextl (nextl Q xs) ys"
  by (induct xs arbitrary: Q) auto

lemma nextl_snoc [simp]: "nextl Q (xs@[x]) = ( $\bigcup$  q  $\in$  nextl Q xs. epsclo
(nxt M q x))"
  by (simp add: nextl_app)

lemma nextl_state: "nextl Q xs  $\subseteq$  states M"
  by (induct xs arbitrary: Q) auto

lemma nextl_mono: "Q'  $\subseteq$  Q  $\implies$  nextl Q' u  $\subseteq$  nextl Q u"
  by (induct u rule: rev_induct) (auto simp: epsclo_mono)

lemma nextl_eps: "q  $\in$  nextl Q u  $\implies$  (q,q')  $\in$  eps M  $\implies$  q'  $\in$  states M
 $\implies$  q'  $\in$  nextl Q u"
  using rtrancl_into_rtrancl epsclo_nextl epsclo_eq_Image by fastforce

lemma finite_nextl: "finite (nextl Q u)"
  by (induct u rule: List.rev_induct) auto

lemma nextl_empty [simp]: "nextl {} xs = {}"
  by (induct xs) auto

lemma nextl_Un: "nextl (Q1  $\cup$  Q2) xs = nextl Q1 xs  $\cup$  nextl Q2 xs"
  by (induct xs arbitrary: Q1 Q2) auto

lemma nextl_UN: "nextl ( $\bigcup$  i  $\in$  I. f i) xs = ( $\bigcup$  i  $\in$  I. nextl (f i) xs)"
  by (induct xs arbitrary: f) auto

end

```

Also works for state type 's of DFA leading to state type 's set in NFA

```

lemma nfa_of_dfa:
  assumes "dfa (M::('a,hf)dfa)"
  obtains N :: "('a,hf)nfa" where "nfa N  $\wedge$  nfa.language N = dfa.language

```

```

M"
proof-
  interpret M: dfa M using assms by blast
  let ?N = "(states = dfa.states M,
            init = {dfa.init M},
            final = dfa.final M,
            nxt =  $\lambda q x. \{dfa.nxt M q x\}$ ,
            eps = {})"
  interpret N: nfa ?N
    using assms dfa_def nfa_def by fastforce
  have " $q \in dfa.states M \longrightarrow N.next1 \{q\} xs = \{dfa.next1 M q xs\}$ " for q
xs
  proof (induction xs arbitrary: q)
    case Cons
    then show ?case by (simp add: M.nxt)
  qed simp
  hence "N.language = M.language"
    by (simp add: N.language_def M.language_def)
  then show ?thesis
    using N.nfa_axioms that[of ?N] by blast
qed

```

```

corollary nfa_if_regular:
  assumes "regular L"
  obtains N :: "('a,hf)nfa" where "nfa N  $\wedge$  nfa.language N = L"
  by (metis assms nfa_of_dfa regular_def)

```

### 1.3.2 The Powerset Construction

First: The construction of a ('a, 's set) dfa from an ('a, 's) nfa. Is not used later on but provides an easy means of showing regularity of some language by constructing an NFA without having to use hf.

```

context nfa
begin

```

```

definition Power_dfa :: "('a,'s set) dfa" where
  "Power_dfa = (dfa.states = { $\text{epscl } q \mid q. q \in \text{Pow } (\text{states } M)\}$ },
            init =  $\text{epscl } (\text{init } M)$ ,
            final = { $\text{epscl } Q \mid Q. Q \subseteq \text{states } M \wedge Q \cap \text{final}$ 
M  $\neq \{\}$ },
            nxt =  $\lambda Q x. \bigcup q \in \text{epscl } Q. \text{epscl } (\text{nxt } M q x)$ )"

```

```

lemma states_Power_dfa [simp]: "dfa.states Power_dfa =  $\text{epscl } ' \text{Pow } (\text{states } M)$ "
by (auto simp add: Power_dfa_def)

```

```

lemma init_Power_dfa [simp]: "dfa.init Power_dfa =  $\text{epscl } (\text{nfa.init } M)$ "
by (simp add: Power_dfa_def)

```

```

lemma final_Power_dfa [simp]: "dfa.final Power_dfa = {eps clo Q | Q. Q
  ⊆ states M ∧ Q ∩ final M ≠ {}}"
  by (simp add: Power_dfa_def)

lemma nxt_Power_dfa [simp]: "dfa.nxt Power_dfa = (λQ x. ⋃ q ∈ eps clo
  Q. eps clo (nxt M q x))"
  by (simp add: Power_dfa_def)

interpretation Power: dfa Power_dfa
proof unfold_locales
  show "dfa.init Power_dfa ∈ dfa.states Power_dfa"
    by (force simp add: init)
next
  show "dfa.final Power_dfa ⊆ dfa.states Power_dfa"
    by auto
next
  fix q a
  show "q ∈ dfa.states Power_dfa ⇒ dfa.nxt Power_dfa q a ∈ dfa.states
  Power_dfa"
    apply (auto simp: nxt)
    apply (metis Pow_iff eps clo_UN eps clo_idem eps clo_subset image_eqI)
    done
next
  show "finite (dfa.states Power_dfa)"
    by (force simp: finite)
qed

```

The Power DFA accepts the same language as the NFA.

```

theorem Power_language [simp]: "Power.language = language"
proof -
  { fix u
    have "(Power.nextl (dfa.init Power_dfa) u) = nextl (init M) u"
    proof (induct u rule: List.rev_induct)
      case Nil show ?case
        using Power.nextl.simps
        by (auto simp: hinsert_def)
    next
      case (snoc x u) then show ?case
        by (simp add: init finite_nextl nextl_state [THEN subsetD])
    qed
    then have "u ∈ Power.language ↔ u ∈ language"
      apply (auto simp add: Power.language_def language_def disjoint_iff_not_equal)
      apply (metis Int_iff eps clo_increasing subsetCE)
      apply (metis eps clo_nextl nextl_state)
      done
  }
  then show ?thesis
    by blast
qed

```

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

```
corollary imp_regular: "regular language"
  by (metis Power.regular_dfa Power_language)
```

end

As above, outside the locale

```
corollary nfa_imp_regular:
  assumes "nfa M" "nfa.language M = L"
  shows "regular L"
using assms nfa_imp_regular by blast
```

```
type_synonym 'a nfa_hf = "('a,hf) nfa"
```

The construction of a 'a dfa\_hf from an 'a nfa\_hf. Very little can be reused from the generic 's-based construction above.

```
locale nfa_hf = nfa M for M :: "'a nfa_hf"
begin
```

```
definition Power_dfa_hf :: "'a dfa_hf" where
  "Power_dfa_hf = ((dfa.states = HF ' dfa.states Power_dfa,
    init = HF (dfa.init Power_dfa),
    final = HF ' dfa.final Power_dfa,
    nxt =  $\lambda Q. HF \circ dfa.nxt \text{ Power\_dfa } (hfset Q)$ )"
```

```
lemma states_Power_dfa [simp]: "dfa.states Power_dfa_hf = HF ' epsclo
  ' Pow (states M)"
  by (auto simp add: Power_dfa_hf_def)
```

```
lemma init_Power_dfa [simp]: "dfa.init Power_dfa_hf = HF (dfa.init Power_dfa)"
  by (simp add: Power_dfa_hf_def)
```

```
lemma final_Power_dfa [simp]: "dfa.final Power_dfa_hf = {HF (eps clo Q)
  | Q. Q  $\subseteq$  states M  $\wedge$  Q  $\cap$  final M  $\neq$  {}}"
  by (auto simp add: Power_dfa_hf_def)
```

```
lemma nxt_Power_dfa [simp]: "dfa.nxt Power_dfa_hf = ( $\lambda Q x. HF(\bigcup q \in$ 
  eps clo (hfset Q). eps clo (nxt M q x)))"
  by (simp add: Power_dfa_hf_def o_def)
```

```
interpretation Power: dfa Power_dfa_hf
```

```
proof
```

```
  fix q x
```

```
  assume "q  $\in$  dfa.states Power_dfa_hf"
```

```
  then show "dfa.nxt Power_dfa_hf q x  $\in$  dfa.states Power_dfa_hf"
```

```
    unfolding dfa_def nxt_Power_dfa
```

```
    by (metis Pow_iff eps clo_UN eps clo_idem eps clo_subset imageI states_Power_dfa)
```

```

qed (use finite init in auto)

corollary dfa_Power: "dfa Power_dfa_hf"
  by unfold_locales

lemma nextl_Power_dfa:
  "qs ∈ dfa.states Power_dfa_hf
  ⇒ dfa.nextl Power_dfa_hf qs u = HF (⋃ q ∈ hfset qs. nextl {q} u)"
proof (induct u rule: List.rev_induct)
  case Nil
  have "⋀ Q. qs = HF (eps clo Q) ⇒ HF (eps clo Q) = HF (⋃ x ∈ eps clo Q.
eps clo {x})"
  by (metis UN_singleton eps clo_UN eps clo_idem)
  with Nil show ?case by auto
next
  case (snoc x xs)
  then show ?case
  by (auto simp: finite_nextl)
qed

```

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

```

lemma Power_right_lang:
  assumes "qs ∈ dfa.states Power_dfa_hf"
  shows "Power.right_lang qs = (⋃ q ∈ hfset qs. right_lang q)"

proof -
  obtain Q where Q: "qs = HF (eps clo Q)" "Q ⊆ nfa.states M"
  using assms by auto
  have "∃ q ∈ hfset qs. l ∈ right_lang q" if "l ∈ Power.right_lang qs" for
  l
  using that Q eps clo_increasing
  unfolding Power.right_lang_def right_lang_def
  by (fastforce simp: nextl_Power_dfa inj_on_HF [THEN inj_on_eq_iff]
finite_nextl)
  moreover have "u ∈ Power.right_lang qs"
  if "q ∈ hfset qs" and "u ∈ right_lang q" for u q
  using that Q
  apply (simp add: Power.right_lang_def right_lang_def)
  apply (rule_tac x="(⋃ x ∈ eps clo Q. nextl {x} u)" in exI)
  using nextl_state apply (auto simp: nextl_Power_dfa)
  done
  ultimately show ?thesis
  by auto
qed

```

The Power DFA accepts the same language as the NFA.

```

theorem Power_language [simp]: "Power.language = language"
proof -

```

```

{ fix u
  have "(Power.nextl (dfa.init Power_dfa_hf) u) = HF (nextl (init M)
u)"
  proof (induct u rule: List.rev_induct)
    case Nil show ?case
      using Power.nextl.simps
      by (auto simp: hinsert_def)
    next
      case (snoc x u) then show ?case
        by (simp add: init finite_nextl nextl_state [THEN subsetD])
  qed
  then have "u ∈ Power.language  $\longleftrightarrow$  u ∈ language"
  apply (auto simp add: Power.language_def language_def disjoint_iff_not_equal)
  apply (metis Int_iff finite_nextl hfset_HF nextl.simps(1) epsclo_increasing
subsetCE)
  apply (metis epsclo_nextl nextl_state)
  done
}
then show ?thesis
  by blast
qed
end

```

## 1.4 Closure Properties for Regular Languages

### 1.4.1 The Empty Language

```

theorem regular_empty: "regular {}"
proof -
  interpret D: dfa_hf "(dfa.states = {0}, init = 0, final = {}, nxt = λq
x. q)"
  by (auto simp: dfa_hf_def dfa_def)
  have "D.language = {}"
  by (simp add: D.language_def)
  then show ?thesis
  unfolding regular_def dfa_def
  by force
qed

```

### 1.4.2 The Empty Word

```

theorem regular_nullstr: "regular {[]}"
proof -
  interpret N: nfa_hf "(states = {0}, init = {0}, final = {0}, nxt = λq
x. {}, eps = {} )"
  by (auto simp: nfa_hf_def nfa_def)
  have " $\bigwedge u. 0 \in N.nextl \{0\} u \implies u = []$ "
  by (rule list.exhaust, auto)

```

```

then have "N.language = {}"
  by (auto simp: N.language_def)
then show ?thesis
  by (metis N.imp_regular)
qed

```

### 1.4.3 Single Symbol Languages

```

theorem regular_singstr: "regular {[a]}"
proof -
  let ?N = "(states = {0,1}, init = {0}, final = {1},
    nxt =  $\lambda q x. \text{if } q=0 \wedge x=a \text{ then } \{1\} \text{ else } \{\}$ ,
    eps = {})" :: 'a nfa_hf"
  interpret N: nfa_hf ?N
    by (auto simp: nfa_def nfa_hf_def)
  have [intro]: " $\bigwedge u. 1 \in N.\text{nextl } \{1\} u \implies u = []$ "
    by (rule list.exhaust) auto
  { fix u
    have " $[1 \in \text{nfa.nextl } ?N \{0\} u] \implies u = [a]$ "
      by (cases u) (auto split: if_split_asm)
    }
  then have "N.language = {[a]}"
    by (auto simp: N.language_def split: if_split_asm)
  then show ?thesis
    by (metis N.imp_regular)
qed

```

### 1.4.4 The Complement of a Language

```

theorem regular_Compl:
  assumes S: "regular S" shows "regular (-S)"
proof -
  obtain MS where M: "dfa_hf MS" and lang: "dfa.language MS = S"
    using S by (auto simp: regular_def dfa_hf_def)
  note M = dfa_hf.axioms[OF M]
  interpret ST: dfa "(dfa.states= dfa.states MS,
    init= dfa.init MS, final= dfa.states MS - dfa.final
MS,
    nxt=  $\lambda q x. \text{dfa.next } MS q x$ )"
    using M
    by (force simp add: dfa_def)
  { fix u
    have "ST.nextl (dfa.init MS) u = dfa.nextl MS (dfa.init MS) u"
      by (induct u rule: List.rev_induct) (auto simp: dfa.nextl_snoc dfa.nextl_simps
M)
    then have "u  $\in$  ST.language  $\longleftrightarrow$  u  $\notin$  dfa.language MS"
      by (auto simp: M dfa.nextl_init_state ST.language_def dfa.language_def)
    }
  then have eq_L: "ST.language = -S"
    using lang by blast

```

```

show ?thesis
  unfolding regular_def
  by (intro exI conjI, unfold_locales) (rule eq_L)
qed

```

### 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 -
  obtain MS MT where M: "dfa_hf MS" "dfa_hf MT" and lang: "dfa.language
MS = S" "dfa.language MT = T"
  using S T by (auto simp: regular_def dfa_hf_def)
  note M = dfa_hf.axioms[OF M(1)] dfa_hf.axioms[OF M(2)]
  interpret ST: dfa "{dfa.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⟩}"
  using M
  by (auto simp: dfa_def finite_image_set2)
  { fix u
    have "ST.nextl ⟨dfa.init MS, dfa.init MT⟩ u =
      ⟨dfa.nextl MS (dfa.init MS) u, dfa.nextl MT (dfa.init MT) u⟩"
    proof (induct u rule: List.rev_induct)
      case Nil show ?case
        by (auto simp: dfa.nextl.simps M)
      next
        case (snoc x u) then show ?case
          by (simp add: dfa.nextl_snoc M)
    qed
    then have "u ∈ ST.language ↔ u ∈ dfa.language MS ∧ u ∈ dfa.language
MT"
      by (auto simp: M ST.language_def dfa.language_def dfa.finite_final)
  }
  then have eq_L: "ST.language = S ∩ T"
    using lang
    by blast
  show ?thesis
    unfolding regular_def
    by (intro exI conjI, unfold_locales) (rule eq_L)
qed

```

```

corollary regular_Un:
  assumes S: "regular S" and T: "regular T" shows "regular (S ∪ T)"
  by (metis S T compl_sup double_compl regular_Cmpl regular_Int [of

```

```
"-S" "-T"])
```

### 1.4.6 The Concatenation of Two Languages

```
lemma Inlr_rtrancl [simp]: "((λq. (HF.Inl q, HF.Inr a)) ' A)* = ((λq.
(HF.Inl q, HF.Inr a)) ' A)=""
  by (auto elim: rtranclE)
```

```
theorem regular_conc:
```

```
  assumes S: "regular S" and T: "regular T" shows "regular (S @@ T)"
proof -
```

```
  obtain MS MT where M: "dfa_hf MS" "dfa_hf MT" and lang: "dfa.language
MS = S" "dfa.language MT = T"
```

```
  using S T by (auto simp: regular_def dfa_hf_def)
```

```
  note M = dfa_hf.axioms[OF M(1)] dfa_hf.axioms[OF M(2)]
```

```
  note [simp] = dfa.init dfa.nxt dfa.nextl.simps dfa.nextl_snoc
```

```
  let ?ST = "(nfa.states = HF.Inl ' (dfa.states MS) ∪ HF.Inr ' (dfa.states
MT),
```

```
    init = {HF.Inl (dfa.init MS)},
```

```
    final = HF.Inr ' (dfa.final MT),
```

```
    nxt = λq x. sum_case (λqs. {HF.Inl (dfa.nxt MS qs
```

```
x}))
```

```
(λqt. {HF.Inr (dfa.nxt MT
```

```
qt x})) q,
```

```
    eps = (λq. (HF.Inl q, HF.Inr (dfa.init MT))) ' dfa.final
```

```
MS))"
```

```
  interpret ST: nfa ?ST
```

```
  using M dfa.final
```

```
  by (force simp add: nfa_def dfa.finite)
```

```
  have Inl_in_eps_iff: "∧q Q. HF.Inl q ∈ nfa.epsclo ?ST Q ↔ HF.Inl
q ∈ Q ∧ q ∈ dfa.states MS"
```

```
  by (auto simp: M dfa.finite ST.epsclo_def)
```

```
  have Inr_in_eps_iff: "∧q Q. HF.Inr q ∈ nfa.epsclo ?ST Q ↔
(HF.Inr q ∈ Q ∧ q ∈ dfa.states MT ∨ (q = dfa.init MT ∧ (∃qf
∈ dfa.final MS. HF.Inl qf ∈ Q)))"
```

```
  by (auto simp: M dfa.finite ST.epsclo_def)
```

```
{ fix u
```

```
  have "∧q. HF.Inl q ∈ ST.nextl {HF.Inl (dfa.init MS)} u ↔ q = (dfa.nextl
MS (dfa.init MS) u)"
```

```
  proof (induct u rule: List.rev_induct)
```

```
    case Nil show ?case
```

```
      by (auto simp: M Inl_in_eps_iff)
```

```
    next
```

```
      case (snoc x u) then show ?case
```

```
        apply (auto simp add: M dfa.nextl_init_state Inl_in_eps_iff is_hsum_def
split: sum_case_split)
```

```
        apply (frule ST.nextl_state [THEN subsetD], auto)
```

```
        done
```

```
      qed
```

```

} note Inl_ST_iff = this
{ fix u
  have "\q. HF.Inr q ∈ ST.nextl {HF.Inl (dfa.init MS)} u ↔
    (∃uS uT. uS ∈ dfa.language MS ∧ u = uS@uT ∧ q = dfa.nextl
MT (dfa.init MT) uT)"
  proof (induct u rule: List.rev_induct)
    case Nil show ?case
      by (auto simp: M dfa.language_def Inr_in_eps_iff)
    next
      case (snoc x u)
      then show ?case using M
        apply (auto simp: Inr_in_eps_iff)
        apply (frule ST.nextl_state [THEN subsetD], force)
        apply (frule ST.nextl_state [THEN subsetD])
        apply (auto simp: dfa.language_def Inl_ST_iff)
        apply (metis (lifting) append_Nil2 dfa.nextl.simps(1) dfa.nextl_snoc)
        apply (rename_tac uS uT)
        apply (rule_tac xs=uT in rev_exhaust, simp)
        apply (rule bexI [where x="HF.Inl (dfa.nextl MS (dfa.init MS)
u)"])
          apply (auto simp: Inl_ST_iff)
          apply (rule bexI)
          apply (auto simp: dfa.nextl_init_state)
          done
        qed
      then have "u ∈ ST.language ↔
        (∃uS uT. uS ∈ dfa.language MS ∧ uT ∈ dfa.language MT ∧
u = uS@uT)"
        by (force simp: M ST.language_def dfa.language_def)
      }
      then have eq_L: "ST.language = S @@ T"
        using lang_unfolding conc_def
        by blast
      then show ?thesis
        using ST.nfa_axioms nfa_hf_def nfa_imp_regular by blast
      qed

lemma regular_word: "regular {u}"
proof (induction u)
  case Nil show ?case
    by (simp add: regular_nullstr)
  next
    case (Cons x l)
    have "{x#l} = {[x]} @@ {l}"
      by (simp add: conc_def)
    then show ?case
      by (simp add: Cons.IH regular_conc regular_singstr)
  qed

```

All finite sets are regular.

```

theorem regular_finite: "finite L  $\implies$  regular L"
  apply (induct L rule: finite.induct)
  apply (simp add: regular_empty)
  using regular_Un regular_word by fastforce

```

### 1.4.7 The Kleene Star of a Language

```

theorem regular_star:
  assumes S: "regular S" shows "regular (star S)"
proof -
  obtain MS where M: "dfa_hf MS" and lang: "dfa.language MS = S"
    using S by (auto simp: regular_def dfa_hf_def)
  note M = dfa_hf.axioms[OF M]
  note [simp] = dfa.init [OF M] dfa.nextl.simps [OF M] dfa.nextl_snoc
    [OF M]
  obtain q0 where q0: "q0  $\notin$  dfa.states MS" using dfa.finite [OF M]
    by (metis hdomain_not_mem hfset_HF hmem_def)
  have [simp]: "q0  $\neq$  dfa.init MS"
    using M dfa.init q0 by blast
  have [simp]: " $\bigwedge q x. q \in \text{dfa.states MS} \implies q0 \neq \text{dfa.nxt MS } q$ "
    using M dfa.nxt q0 by fastforce
  have [simp]: " $\bigwedge q u. q \in \text{dfa.states MS} \implies q0 \neq \text{dfa.nextl MS } q$ "
    using M dfa.nextl_state q0 by metis
  let ?ST = "(nfa.states = insert q0 (dfa.states MS),
    init = {q0},
    final = {q0},
    nxt =  $\lambda q x. \text{if } q \in \text{dfa.states MS} \text{ then } \{\text{dfa.nxt MS } q\}$ 
    else  $\{\}$ ),
    eps = insert (q0, dfa.init MS) (( $\lambda q. (q, q0)$ ) '
    (dfa.final MS)))"
  interpret ST: nfa ?ST
    using M dfa.final
  by (auto simp: q0 nfa_def dfa.init dfa.finite dfa.nxt)
  have " $\bigwedge x y. (x, y) \in (\text{insert } (q0, \text{dfa.init MS}) ((\lambda q. (q, q0)) ' \text{dfa.final MS}))^* \iff$ 
    ( $x=y$ )  $\vee$  ( $x = q0 \wedge y = \text{dfa.init MS}$ )  $\vee$ 
    ( $x \in \text{dfa.final MS} \wedge y \in \{q0, \text{dfa.init MS}\}$ )"
  apply (rule iffI)
  apply (erule rtrancl_induct)
  apply (auto intro: rtrancl.rtrancl_into_rtrancl)
  done
  then have eps_iff:
    " $\bigwedge q Q. q \in \text{ST.epscl} Q \iff$ 
     $q \in Q \cap \text{insert } q0 (\text{dfa.states MS}) \vee$ 
    ( $q = q0 \wedge \text{dfa.final MS} \cap Q \neq \{\}$ )  $\vee$ 
    ( $q = \text{dfa.init MS} \wedge \text{insert } q0 (\text{dfa.final MS}) \cap Q \neq \{\}$ )"
  by (auto simp: q0 ST.epscl_def)
  { fix u
    have "dfa.nextl MS (dfa.init MS) u  $\in$  ST.nextl {q0} u"

```

```

proof (induct u rule: List.rev_induct)
  case Nil show ?case
    by (auto simp: M eps_iff)
next
  case (snoc x u) then show ?case
    using M q0
    apply (simp add: eps_iff)
    apply (rule bexI [where x="dfa.nextl MS (dfa.init MS) u"])
    apply (auto simp: dfa.nextl_init_state dfa.nxt)
    done
qed
} note dfa_in_ST = this
{ fix ustar
  assume "ustar ∈ star (dfa.language MS)"
  then have "q0 ∈ ST.nextl {q0} ustar"
  proof (induct rule: star_induct)
    case Nil show ?case
      by (auto simp: eps_iff)
  next
    case (append u v)
    then have "dfa.nextl MS (dfa.init MS) u ∈ dfa.final MS"
      by (simp add: M dfa.language_def)
    then have "q0 ∈ ST.nextl {q0} u"
      by (auto intro: ST.nextl_eps dfa_in_ST)
    then show ?case
      using append ST.nextl_mono
      by (clarsimp simp add: ST.nextl_app) blast
  qed
} note star_dfa_in_ST = this
{ fix u q
  assume "q ∈ ST.nextl {q0} u"
  then have "q = q0 ∧ u=[] ∨
    (∃u1 u2. u = u1@u2 ∧ u1 ∈ star (dfa.language MS) ∧
    (q ∈ ST.epscl {dfa.nextl MS (dfa.init MS) u2}))"
  proof (induction u arbitrary: q rule: List.rev_induct)
    case Nil then show ?case
      by (auto simp: M dfa.init eps_iff)
  next
    case (snoc x u) show ?case using snoc.prem q0 M
      apply (auto split: if_split_asm simp: dfa.language_def eps_iff
        dest!: snoc.IH)
      apply (metis dfa.nextl_snoc)
      apply (rename_tac u1 u2)
      apply (rule_tac x="u1@u2" in exI, auto)
      apply (rule_tac x = u1 in exI, auto)
      apply (rule_tac x="u1@u2" in exI, auto)
      apply (rule_tac x="u1@u2@[x]" in exI, auto)
      apply (rule_tac x="u1@u2@[x]" in exI, auto)
      done
  }

```

```

    qed
  } note in_ST_imp = this
  have "\u. q0 \in ST.nextl {q0} u \implies u \in star (dfa.language MS)"
    by (auto simp: M dfa.init dfa.language_def eps_iff dest: in_ST_imp)
  then have eq_L: "ST.language = star S"
    using lang
    by (auto simp: ST.language_def star_dfa_in_ST)
  then show ?thesis
    using ST.nfa_axioms nfa_hf.intro nfa_imp_regular by blast
qed

```

### 1.4.8 The Reversal of a Regular Language

```

definition Reverse_nfa :: "'a dfa_hf \implies 'a nfa_hf" where
  "Reverse_nfa MS = (nfa.states = dfa.states MS,
    init = dfa.final MS,
    final = {dfa.init MS},
    nxt = \q x. {q' \in dfa.states MS. q = dfa.nxt
MS q' x},
    eps = {})"

```

```

lemma states_Reverse_nfa [simp]: "states (Reverse_nfa MS) = dfa.states MS"
  by (simp add: Reverse_nfa_def)

```

```

lemma init_Reverse_nfa [simp]: "init (Reverse_nfa MS) = dfa.final MS"
  by (simp add: Reverse_nfa_def)

```

```

lemma final_Reverse_nfa [simp]: "final (Reverse_nfa MS) = {dfa.init MS}"
  by (simp add: Reverse_nfa_def)

```

```

lemma nxt_Reverse_nfa [simp]:
  "nxt (Reverse_nfa MS) q x = {q' \in dfa.states MS. q = dfa.nxt MS q' x}"
  by (simp add: Reverse_nfa_def)

```

```

lemma eps_Reverse_nfa [simp]: "eps (Reverse_nfa MS) = {}"
  by (simp add: Reverse_nfa_def)

```

```

context dfa_hf
begin

```

```

  lemma nfa_Reverse_nfa: "nfa (Reverse_nfa M)"
    by unfold_locales (auto simp: final finite)

```

```

  lemma nextl_Reverse_nfa:
    "nfa.nextl (Reverse_nfa M) Q u = {q' \in dfa.states M. dfa.nextl M q'
(rev u) \in Q}"
  proof -
    interpret NR: nfa "Reverse_nfa M"

```

```

    by (rule nfa_Reverse_nfa)
  show ?thesis
    by (induct u rule: rev_induct) (auto simp: nxt)
qed

```

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 -
  interpret NR: nfa "Reverse_nfa M"
    by (rule nfa_Reverse_nfa)
  show ?thesis
    by (force simp add: left_lang_def NR.right_lang_def nfa_Reverse_nfa
nextl_Reverse_nfa)
qed

```

```

lemma right_lang_Reverse_disjoint:
  "q1 ≠ q2 ⇒ nfa.right_lang (Reverse_nfa M) q1 ∩ nfa.right_lang (Reverse_nfa
M) q2 = {}"
  by (auto simp: left_lang_def right_lang_Reverse)

```

```

lemma epsclo_Reverse_nfa [simp]: "nfa.epsclo (Reverse_nfa M) Q = Q
∩ dfa.states M"
  by (auto simp: nfa.epsclo_def nfa_Reverse_nfa)

```

```

theorem language_Reverse_nfa [simp]:
  "nfa.language (Reverse_nfa M) = (rev ' dfa.language M)"
proof -
  interpret NR: nfa "Reverse_nfa M"
    by (rule nfa_Reverse_nfa)
  have "NR.language = {u. rev u ∈ dfa.language M}"
  proof (rule set_eqI, simp)
    fix u
    have "∧Q q'. q' ∈ dfa.states M ⇒
      q' ∈ NR.nextl Q u ↔ dfa.nextl M q' (rev u) ∈ Q"
      by (induct u rule: List.rev_induct) (auto simp: nxt)
    then show "u ∈ nfa.language (Reverse_nfa M) ↔ rev u ∈ language"
      by (simp add: NR.language_def language_def)
  qed
  then show ?thesis
    by (force simp add: language_def)
qed

```

end

```

corollary regular_Reverse:
  assumes S: "regular S" shows "regular (rev ' S)"
proof -

```

```

obtain MS where MS: "dfa_hf MS" "dfa.language MS = S"
  using S by (auto simp: regular_def dfa_hf_def)
then interpret dfa_hf "MS"
  by simp
show ?thesis
  by (metis MS(2) language_Reverse_nfa nfa_Reverse_nfa nfa_imp_regular)
qed

```

All regular expressions yield regular languages.

```

theorem regular_lang: "regular (lang r)"
  by (induct r)
  (auto simp: regular_empty regular_nullstr regular_singstr regular_Un
regular_conc regular_star)

```

## 1.5 Brzowski's Minimization Algorithm

```

context dfa_hf
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"
  by (simp add: eq_app_right_def left_lang_def language_def nextl_app)

```

```

lemma eq_app_right_class_eq:
  "UNIV // eq_app_right language = (λq. eq_app_right language ‘‘ {path_to
q}) ‘ accessible"
proof -
  { fix u
    have "eq_app_right language ‘‘ {u} ∈ (λq. eq_app_right language
‘‘ {path_to q}) ‘ accessible"
      apply (rule image_eqI)
      apply (rule equiv_class_eq [OF equiv_eq_app_right])
      apply (rule left_eq_app_right [OF _ path_to_left_lang])
      apply (auto simp: left_lang_def accessible_def)
      done
  }
  then show ?thesis
    by (auto simp: quotient_def)
qed

```

```

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)"
using assms
apply (auto simp: bij_betw_def inj_on_def eq_app_right_class_eq)
apply (drule eq_equiv_class [OF _ equiv_eq_app_right])

```

```

apply (auto simp: nextl_path_to eq_app_right_def language_def right_lang_def
         nextl_app accessible_imp_states)
done

```

```

definition min_states where
  "min_states  $\equiv$  card (UNIV // eq_app_right language)"

```

```

lemma minimal_imp_index_eq_app_right:
  "minimal  $\implies$  card (dfa.states M) = min_states"
  unfolding min_states_def minimal_def
  by (metis bij_betw_def card_image inj_right_lang_imp_eq_app_right_index)

```

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]]
    $\implies$  card (dfa.states M)  $\leq$  card (dfa.states M')]"
  using minimal_imp_index_eq_app_right dfa.index_eq_app_right_lower
  min_states_def
  by fastforce

```

```

definition index_f :: "'a list set  $\Rightarrow$  hf" where
  "index_f  $\equiv$  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 -
    have " $\exists$ h. bij_betw h (UNIV // eq_app_right language) (hfset (ord_of
  min_states))"
      unfolding min_states_def
      by (metis L2_3 MN_eq_nextl ex_bij_betw_finite_nat bij_betw_ord_ofI)
    then show ?thesis
      unfolding index_f_def by (metis someI_ex)
  qed

```

```

interpretation Canon:
  MyhillNerode_dfa language "eq_app_right language"
  language
  min_states index_f
  by (simp add: MyhillNerode_dfa_def equiv_eq_app_right right_invariant_eq_app_right
  index_f eq_app_right_eq)

```

```

interpretation MN: dfa Canon.DFA
  by (fact Canon.dfa)

```

```

definition iso :: "hf  $\Rightarrow$  hf" where
  "iso  $\equiv$  index_f o ( $\lambda$ q. eq_app_right language ‘‘ {path_to q})"

```

```

theorem minimal_imp_isomorphic_to_canonical:
  assumes minimal
  shows "dfa_isomorphism M Canon.DFA iso"
proof (unfold_locales, simp_all add: Canon.DFA_def)
  have "bij_betw iso accessible (hfset (ord_of min_states))"
    using assms bij_betw_trans index_f inj_right_lang_imp_eq_app_right_index
    unfolding iso_def minimal_def
    by blast
  then show "bij_betw iso (dfa.states M) (index_f ' (UNIV // eq_app_right
language))"
    by (metis assms bij_betw_def index_f minimal_def)
  next
  have "eq_app_right language ' ' {path_to (dfa.init M)} = eq_app_right
language ' ' {}"
  proof (rule equiv_class_eq [OF equiv_eq_app_right])
    show "(path_to (dfa.init M), []) ∈ eq_app_right language"
      using nextl_path_to assms
      by (auto simp: minimal_def eq_app_right_def language_def nextl_app)
    qed
  then show "iso (dfa.init M) = index_f (eq_app_right language ' '
{})"
    by (simp add: iso_def)
  next
  have "(λu. eq_app_right language ' ' {path_to u}) ' dfa.final M =
(λl. eq_app_right language ' ' {l}) ' language"
    using assms final nextl_path_to nextl_app
    apply (auto simp: dfa_isomorphism_def language_def minimal_def)
    apply (auto simp: eq_app_right_def
      intro: rev_image_eqI equiv_class_eq [OF equiv_eq_app_right])
    done
  from this [THEN image_eq_imp_comp [where h = index_f]]
  show "iso ' dfa.final M = {index_f (eq_app_right language ' ' {u})
|u. u ∈ language}"
    by (simp add: iso_def o_def Setcompr_eq_image)
  next
  have nxt: "∧q x. q ∈ dfa.states M ⇒
eq_app_right language ' ' {path_to (dfa.nxt M q
x)} =
eq_app_right language ' ' {path_to q @ [x]}"
  apply (rule equiv_class_eq [OF equiv_eq_app_right])
  using assms nextl_path_to nxt nextl_app
  apply (auto simp: minimal_def nextl_path_to eq_app_right_def language_def)
  done
  show "∧q x. q ∈ dfa.states M ⇒
iso (dfa.nxt M q x) =
index_f
(∪u∈MyhillNerode_dfa.hinv (eq_app_right language) index_f
(iso q).
eq_app_right language ' ' {u @ [x]})"

```

```

      by (simp add: iso_def Canon.injh Canon.hinv_def quotientI Canon.resp
nxt
          UN_equiv_class [OF equiv_eq_app_right])
qed

lemma states_PR [simp]:
  "dfa.states (nfa_hf.Power_dfa_hf (Reverse_nfa M)) = HF ' Pow (dfa.states
M)"
  by (rule set_eqI)
  (auto simp: nfa_hf.states_Power_dfa nfa_Reverse_nfa nfa_hf.intro
Bex_def)

lemma inj_on_right_lang_PR:
  assumes "dfa.states M = accessible"
  shows "inj_on (dfa.right_lang (nfa_hf.Power_dfa_hf (Reverse_nfa
M)))
          (dfa.states (nfa_hf.Power_dfa_hf (Reverse_nfa M)))"
proof (rule inj_onI)
  fix q1 q2
  assume *: "q1 ∈ dfa.states (nfa_hf.Power_dfa_hf (Reverse_nfa M))"
          "q2 ∈ dfa.states (nfa_hf.Power_dfa_hf (Reverse_nfa M))"
          "dfa.right_lang (nfa_hf.Power_dfa_hf (Reverse_nfa M)) q1
=
          dfa.right_lang (nfa_hf.Power_dfa_hf (Reverse_nfa M)) q2"
  then have "hfset q1 ⊆ accessible ∧ hfset q2 ⊆ accessible"
    using assms rev_finite_subset [OF finite]
    by force
  with * show "q1 = q2"
    apply (simp add: nfa_Reverse_nfa nfa_hf.Power_right_lang right_lang_Reverse
image_UN [symmetric] inj_image_eq_iff nfa_hf_def)
    apply (metis HF_hfset le_sup_iff left_lang_UN)
    done
qed

abbreviation APR :: "'x dfa_hf ⇒ 'x dfa_hf" where
  "APR X ≡ dfa.Accessible_dfa (nfa_hf.Power_dfa_hf (Reverse_nfa X))"

theorem minimal_APR:
  assumes "dfa.states M = accessible"
  shows "dfa.minimal (APR M)"
proof -
  have PR: "dfa (APR M)"
    "dfa (nfa_hf.Power_dfa_hf (Reverse_nfa M))"
  by (auto simp add: dfa.dfa_Accessible nfa_Reverse_nfa nfa_hf.dfa_Power
nfa_hf.intro)
  then show ?thesis
    apply (simp add: dfa.minimal_def dfa.states_Accessible_dfa dfa.Accessible_accessible)
    apply (simp add: inj_on_def dfa.Accessible_right_lang_eq)
    apply (meson assms dfa.accessible_imp_states inj_onD inj_on_right_lang_PR)

```

```

    done
  qed

definition Brzozowski :: "'a dfa_hf" where
  "Brzozowski  $\equiv$  APR (APR M)"

lemma dfa_Brzozowski: "dfa_hf Brzozowski"
  by (simp add: Brzozowski_def dfa.dfa_Accessible dfa_hf.nfa_Reverse_nfa
    nfa_hf.dfa_Power nfa_Reverse_nfa dfa_hf_def nfa_hf_def)

theorem language_Brzozowski: "dfa.language Brzozowski = language"
  by (simp add: Brzozowski_def dfa.Accessible_language nfa_hf.Power_language
    dfa.dfa_Accessible dfa_hf.nfa_Reverse_nfa nfa_hf.dfa_Power nfa_Reverse_nfa
    dfa_hf.language_Reverse_nfa image_image dfa_hf_def nfa_hf_def)

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

lemma index_f_cong:
  "[[dfa.language M = dfa.language N; dfa M; dfa N]]  $\implies$  dfa_hf.index_f
  M = dfa_hf.index_f N"
  by (simp add: dfa_hf.index_f_def dfa_hf.min_states_def dfa_hf_def)

theorem minimal_imp_isomorphic:
  "[[dfa.language M = dfa.language N; dfa.minimal M; dfa.minimal N;
  dfa_hf M; dfa_hf N]]
   $\implies$   $\exists$ h. dfa_isomorphism M N h"
  by (metis dfa_hf.minimal_imp_isomorphic_to_canonical dfa_hf_def dfa_isomorphism.sym
    dfa_isomorphism.trans index_f_cong)

end

```