# Automata.thy

```
(* This file can be checked using any recent version of the
Isabelle proof assistant: http://isabelle.in.tum.de/ *)

theory Automata imports Main begin

(*
The usual definition of a DFA or NFA is a 5-tuple (Q,Σ,q₀,δ,F).
Here, since we are using a typed logic (all values are assigned a type)
we can represent Q and Σ as types. A DFA or NFA can be written with
just the three components (q₀,δ,F); Q and Σ are inferred from those.
*)

types ('Q,'Σ) dfa = "'Q × ('Q × 'Σ ⇒ 'Q) × 'Q set";

types ('Q,'Σ) nfa = "'Q × ('Q × 'Σ ⇒ 'Q set) × 'Q set";


(* Language of a DFA *)
(*
The book definition says for w = w1 w2 ... wn, there exists
a sequence of states r = r0, r1, r2, ... rn such that for
all i≤n, r(i+1) = δ(r(i), w(i+1)). This definition looks a
little different because the list-indexing operator (w!i)
is zero-based. So (w!0) actually means the first symbol in w,
which the book calls w1, and (w!i) stands for w(i+1).
*)

definition dL :: "('Q,'Σ) dfa ⇒ ('Σ list) set" where
  "dL ≡ λ(q₀, δ, F).
    {w. ∃r. (length r = length w + 1)
          ∧ (r!0 = q₀)
          ∧ (∀i. i < length w ⟶ r!(i+1) = δ(r!i, w!i))
          ∧ (r!(length w) ∈ F) }";


(* Language of an NFA *)

definition nL :: "('Q,'Σ) nfa ⇒ ('Σ list) set" where
  "nL ≡ λ(q₀, δ, F).
    {w. ∃r. (length r = length w + 1)
          ∧ (r!0 = q₀)
          ∧ (∀i. i < length w ⟶ r!(i+1) ∈ δ(r!i, w!i))
          ∧ (r!(length w) ∈ F) }";


(* δ-hat functions for DFAs and NFAs *)
(*
About notation: [] denotes the empty string ε.
(a # w) denotes a string (of type 'Σ list) with the
symbol a (of type 'Σ) stuck on the front of the string
w (of type 'Σ list).
*)
```

```
fun dHat :: "('Q × 'Σ ⇒ 'Q) ⇒ ('Q × 'Σ list ⇒ 'Q)" where
  "dHat δ (q, []) = q"
| "dHat δ (q, a # w) = dHat δ (δ(q,a), w)"

fun nHat :: "('Q × 'Σ ⇒ 'Q set) ⇒ ('Q × 'Σ list ⇒ 'Q set)" where
  "nHat δ (q, []) = {q}"
| "nHat δ (q, a#w) = (⋃r∈(δ(q,a)). nHat δ (r, w))";



(*
Four preliminary lemmas with uninteresting proofs:
Language membership for list constructors
*)

lemma dL_Nil: "[] ∈ dL (q0, δ, F) ⟷ q0 ∈ F"
apply (simp add: dL_def)
apply (rule iffI)
apply clarsimp
apply (rule_tac x="[q0]" in exI, simp)
done;

lemma dL_Cons: "w1 # w ∈ dL (q0, δ, F) ⟷ w ∈ dL (δ(q0, w1), δ, F)"
apply (simp add: dL_def)
apply (rule iffI)
apply clarify
apply (case_tac r)
apply simp
apply (rule_tac x="list" in exI)
apply simp
apply clarify
apply (rule_tac x="q0 # r" in exI)
apply simp
apply clarify
apply (case_tac i, simp_all)
done;

lemma nL_Nil: "[] ∈ nL (q0, δ, F) ⟷ q0 ∈ F"
apply (simp add: nL_def)
apply (rule iffI)
apply clarsimp
apply (rule_tac x="[q0]" in exI, simp)
done;

lemma nL_Cons:
  "w1 # w ∈ nL (q0, δ, F) ⟷ (∃q1∈δ(q0, w1). w ∈ nL (q1, δ, F))"
apply (simp add: nL_def)
apply (rule iffI)
apply clarify
apply (rule_tac x="r!1" in rev_bexI)
apply (drule_tac x="0" in spec, simp)
apply (case_tac r)
apply simp
```

```
apply (rule_tac x="list" in exI)
apply simp
apply clarify
apply (drule_tac x="Suc i" in spec)
apply simp
apply clarify
apply (rule_tac x="q0 # r" in exI)
apply simp
apply clarify
apply (case_tac i)
apply simp
apply simp
done;



(* Language membership in terms of δ-hat functions *)
(*
The next two rules are like alternative definitions for dL and nL.
The δ-hat-based definitions are useful for inductive proofs, while
the original definitions are more useful for non-inductive proofs.
*)

lemma dL_dHat:
  "∀q0. w ∈ dL (q0, δ, F) ⟷ dHat δ (q0, w) ∈ F"
by (induct w, simp_all add: dL_Nil dL_Cons);

lemma nL_nHat:
  "∀q0. w ∈ nL (q0, δ, F) ⟷ (∃r ∈ nHat δ (q0, w). r ∈ F)"
by (induct w, simp_all add: nL_Nil nL_Cons);



(*

For any NFA M, we can construct a DFA M' with the same language.

First we want to prove that w ∈ L(M) ⟶ w ∈ L(M').

Start with a lemma about δ-hat functions.

*)

(* First try: *)

lemma nHat_dHat:
  assumes δ': "∀R a. δ'(R, a) = (⋃r∈R. δ(r, a))"

  shows "q ∈ nHat δ (p, w) ⟶ q ∈ dHat δ' ({p}, w)"

apply (induct w)
(*
goal (2 subgoals):
 1. q ∈ nHat δ (p, []) ⟶ q ∈ dHat δ' ({p}, [])
```

```
 2. ⋀a w. q ∈ nHat δ (p, w) ⟶ q ∈ dHat δ' ({p}, w) ⟹
         q ∈ nHat δ (p, a # w) ⟶ q ∈ dHat δ' ({p}, a # w)
*)
apply (simp only: nHat.simps dHat.simps)
(*
 1. q ∈ {p} ⟶ q ∈ {p}
*)
apply simp (* first goal is solved *)
apply (simp only: nHat.simps dHat.simps)
(*
goal (1 subgoal):
 1. ⋀a w. q ∈ nHat δ (p, w) ⟶ q ∈ dHat δ' ({p}, w) ⟹
       q ∈ (⋃r∈δ (p, a). nHat δ (r, w)) ⟶ q ∈ dHat δ' (δ' ({p}, a),
w)
*)
apply simp
(*
 1. ⋀a w. q ∈ nHat δ (p, w) ⟶ q ∈ dHat δ' ({p}, w) ⟹
       (∃r∈δ (p, a). q ∈ nHat δ (r, w)) ⟶ q ∈ dHat δ' (δ' ({p}, a), w)
*)
apply (simp add: δ')
(*
 1. ⋀a w. q ∈ nHat δ (p, w) ⟶ q ∈ dHat δ' ({p}, w) ⟹
       (∃r∈δ (p, a). q ∈ nHat δ (r, w)) ⟶ q ∈ dHat δ' (δ (p, a), w)
*)
(* inductive hypothesis doesn't match the goal *)
oops;


(* Second try: *)
(* Generalize from {p} to arbitrary set S containing p *)

lemma nHat_dHat:
  assumes δ': "∀R a. δ'(R, a) = (⋃r∈R. δ(r, a))"

  shows "∀S. q ∈ nHat δ (p, w) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, w)"

proof (induct w)
  case Nil
  show "∀S. q ∈ nHat δ (p, []) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, [])"
  apply (simp only: nHat.simps dHat.simps)
  (*
  goal (1 subgoal):
   1. ∀S. q ∈ {p} ⟶ p ∈ S ⟶ q ∈ S
  *)
  apply simp
  done
next
  case (Cons a w)
  assume IH: "∀S. q ∈ nHat δ (p, w) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, w)"
  show "∀S. q ∈ nHat δ (p, a#w) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, a#w)"
  apply (simp only: nHat.simps dHat.simps)
  (*
  goal (1 subgoal):
```

```
   1. ∀S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w)) ⟶
          p ∈ S ⟶ q ∈ dHat δ' (δ' (S, a), w)
  *)
  apply clarify
  (*
  goal (1 subgoal):
   1. ⋀S r. ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧
            ⟹ q ∈ dHat δ' (δ' (S, a), w)
  *)
  apply (rule IH [rule_format])
  (*
  goal (2 subgoals):
   1. ⋀S r. ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧
            ⟹ q ∈ nHat δ (p, w)
   2. ⋀S r. ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧
            ⟹ p ∈ δ' (S, a)
  *)
  (* premise of inductive hypothesis doesn't match assumptions *)
  oops;


(* Third try: *)
(* Universally quantify over state p *)

lemma nHat_dHat:
  assumes δ': "∀R a. δ'(R, a) = (⋃r∈R. δ(r, a))"

  shows "∀p S. q ∈ nHat δ (p, w) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, w)"

proof (induct w)
  case Nil
  show "∀p S. q ∈ nHat δ (p, []) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, [])"
  apply (simp only: nHat.simps dHat.simps)
  (*
  goal (1 subgoal):
   1. ∀p S. q ∈ {p} ⟶ p ∈ S ⟶ q ∈ S
  *)
  apply simp
  done
next
  case (Cons a w)
  assume IH: "∀p S. q ∈ nHat δ (p, w) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, w)"
  show "∀p S. q ∈ nHat δ (p, a#w) ⟶ p ∈ S ⟶ q ∈ dHat δ' (S, a#w)"
  apply (simp only: nHat.simps dHat.simps)
  (*
  goal (1 subgoal):
   1. ∀p S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w)) ⟶
            p ∈ S ⟶ q ∈ dHat δ' (δ' (S, a), w)
  *)
  apply clarify
  (*
  goal (1 subgoal):
   1. ⋀p S r.
        ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧
```

```
        ⟹ q ∈ dHat δ' (δ' (S, a), w)
  *)
  apply (rule_tac p="r" and S="δ'(S, a)" in IH [rule_format])
  (*
  goal (2 subgoals):
   1. ⋀p S r.
        ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧
        ⟹ q ∈ nHat δ (r, w)
   2. ⋀p S r.
        ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧ ⟹ r ∈ δ' (S, a)
  *)
  apply assumption
  (*
  goal (1 subgoal):
   1. ⋀p S r.
        ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧ ⟹ r ∈ δ' (S, a)
  *)
  apply (simp only: δ')
  (*
  goal (1 subgoal):
   1. ⋀p S r.
        ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧
        ⟹ r ∈ (⋃r∈S. δ (r, a))
  *)
  apply (rule_tac a="p" in UN_I)
  (*
  goal (2 subgoals):
   1. ⋀p S r. ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧ ⟹ p ∈ S
   2. ⋀p S r.
        ⟦r ∈ δ (p, a); q ∈ nHat δ (r, w); p ∈ S⟧ ⟹ r ∈ δ (p, a)
  *)
  apply assumption
  apply assumption
  done
qed;


(* Now we can prove that w ∈ L(M) ⟶ w ∈ L(M') *)

lemma nL_dL:
  fixes M :: "('Q, 'Σ) nfa" and M' :: "('Q set, 'Σ) dfa"
  assumes M: "M = (q0, δ, F)"
  assumes M': "M' = ({q0}, δ', F')"
  assumes δ': "∀R a. δ'(R, a) = (⋃r∈R. δ(r, a))"
  assumes F': "F' = {R. ∃r∈R. r ∈ F}"
  shows "w ∈ nL(M) ⟶ w ∈ dL(M')"
apply (simp only: M M' nL_nHat dL_dHat)
(*
goal (1 subgoal):
 1. (∃r∈nHat δ (q0, w). r ∈ F) ⟶ dHat δ' ({q0}, w) ∈ F'
*)
apply clarify
(*
goal (1 subgoal):
```

```
 1. ⋀r. ⟦r ∈ nHat δ (q0, w); r ∈ F⟧ ⟹ dHat δ' ({q0}, w) ∈ F'
*)
apply (simp add: F')
(*
goal (1 subgoal):
 1. ⋀r. ⟦r ∈ nHat δ (q0, w); r ∈ F⟧ ⟹ ∃r∈dHat δ' ({q0}, w). r ∈ F
*)
apply (rule_tac x="r" in bexI)
(*
goal (2 subgoals):
 1. ⋀r. ⟦r ∈ nHat δ (q0, w); r ∈ F⟧ ⟹ r ∈ F
 2. ⋀r. ⟦r ∈ nHat δ (q0, w); r ∈ F⟧ ⟹ r ∈ dHat δ' ({q0}, w)
*)
apply assumption
(*
goal (1 subgoal):
 1. ⋀r. ⟦r ∈ nHat δ (q0, w); r ∈ F⟧ ⟹ r ∈ dHat δ' ({q0}, w)
*)
apply (rule_tac p="q0" in nHat_dHat [OF δ', rule_format])
(*
goal (2 subgoals):
 1. ⋀r. ⟦r ∈ nHat δ (q0, w); r ∈ F⟧ ⟹ r ∈ nHat δ (q0, w)
 2. ⋀r. ⟦r ∈ nHat δ (q0, w); r ∈ F⟧ ⟹ q0 ∈ {q0}
*)
apply assumption
apply simp
done;



(* Now we want to show the converse: w ∈ L(M') ⟶ w ∈ L(M) *)
(* Start with a lemma about δ-hat functions *)

lemma dHat_nHat:
  assumes δ': "∀R a. δ'(R, a) = (⋃r∈R. δ(r, a))"

  shows "∀S. q ∈ dHat δ' (S, w) ⟶ (∃p∈S. q ∈ nHat δ (p, w))"

proof (induct w)
  case Nil
  show "∀S. q ∈ dHat δ' (S, []) ⟶ (∃p∈S. q ∈ nHat δ (p, []))"
  apply (simp only: nHat.simps dHat.simps)
  (*
  goal (1 subgoal):
   1. ∀S. q ∈ S ⟶ (∃p∈S. q ∈ {p})
  *)
  apply simp
  done
next
  case (Cons a w)
  assume IH: "∀S. q ∈ dHat δ' (S, w) ⟶ (∃p∈S. q ∈ nHat δ (p, w))"
  show "∀S. q ∈ dHat δ' (S, a#w) ⟶ (∃p∈S. q ∈ nHat δ (p, a#w))"
  apply (simp only: nHat.simps dHat.simps)
```

```
(*
goal (1 subgoal):
 1. ∀S. q ∈ dHat δ' (δ' (S, a), w) ⟶
       (∃p∈S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w)))
*)
apply clarify
(*
goal (1 subgoal):
 1. ⋀S. q ∈ dHat δ' (δ' (S, a), w) ⟹
       ∃p∈S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w))
*)
apply (drule_tac IH [rule_format])
(*
goal (1 subgoal):
 1. ⋀S. ∃p∈δ' (S, a). q ∈ nHat δ (p, w) ⟹
       ∃p∈S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w))
*)
apply clarify
(*
goal (1 subgoal):
 1. ⋀S p. ⟦p ∈ δ' (S, a); q ∈ nHat δ (p, w)⟧
          ⟹ ∃p∈S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w))
*)
apply (simp only: δ')
(*
goal (1 subgoal):
 1. ⋀S p. ⟦p ∈ (⋃r∈S. δ (r, a)); q ∈ nHat δ (p, w)⟧
          ⟹ ∃p∈S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w))
*)
apply clarify
(*
goal (1 subgoal):
 1. ⋀S p r.
       ⟦q ∈ nHat δ (p, w); r ∈ S; p ∈ δ (r, a)⟧
       ⟹ ∃p∈S. q ∈ (⋃r∈δ (p, a). nHat δ (r, w))
*)
apply (rule_tac x="r" in rev_bexI)
(*
goal (2 subgoals):
 1. ⋀S p r. ⟦q ∈ nHat δ (p, w); r ∈ S; p ∈ δ (r, a)⟧ ⟹ r ∈ S
 2. ⋀S p r.
       ⟦q ∈ nHat δ (p, w); r ∈ S; p ∈ δ (r, a)⟧
       ⟹ q ∈ (⋃r∈δ (r, a). nHat δ (r, w))
*)
apply assumption
(*
goal (1 subgoal):
 1. ⋀S p r.
       ⟦q ∈ nHat δ (p, w); r ∈ S; p ∈ δ (r, a)⟧
       ⟹ q ∈ (⋃r∈δ (r, a). nHat δ (r, w))
*)
apply (rule_tac a="p" in UN_I)
(*
goal (2 subgoals):
```

```
   1. ⋀S p r.
         ⟦q ∈ nHat δ (p, w); r ∈ S; p ∈ δ (r, a)⟧
         ⟹ p ∈ δ (r, a)
   2. ⋀S p r.
         ⟦q ∈ nHat δ (p, w); r ∈ S; p ∈ δ (r, a)⟧
         ⟹ q ∈ nHat δ (p, w)
  *)
  apply assumption
  apply assumption
  done
qed;


(* Use the lemma to show w ∈ L(M') ⟶ w ∈ L(M) *)

lemma dL_nL:
  fixes M :: "('Q, 'Σ) nfa" and M' :: "('Q set, 'Σ) dfa"
  assumes M: "M = (q0, δ, F)"
  assumes M': "M' = ({q0}, δ', F')"
  assumes δ': "∀R a. δ'(R, a) = (⋃r∈R. δ(r, a))"
  assumes F': "F' = {R. ∃r∈R. r ∈ F}"
  shows "w ∈ dL(M') ⟶ w ∈ nL(M)"
apply (simp only: M M' dL_dHat nL_nHat)
(*
goal (1 subgoal):
 1. dHat δ' ({q0}, w) ∈ F' ⟶ (∃r∈nHat δ (q0, w). r ∈ F)
*)
apply (simp add: F')
(*
goal (1 subgoal):
 1. (∃r∈dHat δ' ({q0}, w). r ∈ F) ⟶
    (∃r∈nHat δ (q0, w). r ∈ F)
*)
apply clarify
(*
goal (1 subgoal):
 1. ⋀r. ⟦r ∈ dHat δ' ({q0}, w); r ∈ F⟧
        ⟹ ∃r∈nHat δ (q0, w). r ∈ F
*)
apply (drule dHat_nHat [OF δ', rule_format])
(*
goal (1 subgoal):
 1. ⋀r. ⟦r ∈ F; ∃p∈{q0}. r ∈ nHat δ (p, w)⟧
        ⟹ ∃r∈nHat δ (q0, w). r ∈ F
*)
apply (erule bexE)
(*
goal (1 subgoal):
 1. ⋀r p. ⟦r ∈ F; p ∈ {q0}; r ∈ nHat δ (p, w)⟧
          ⟹ ∃r∈nHat δ (q0, w). r ∈ F
*)
apply simp
(*
goal (1 subgoal):
```

```
 1. ⋀r p. ⟦r ∈ F; p = q₀; r ∈ nHat δ (q₀, w)⟧
          ⟹ ∃r∈nHat δ (q₀, w). r ∈ F
*)
apply (rule_tac x="r" in bexI)
(*
goal (2 subgoals):
 1. ⋀r p. ⟦r ∈ F; p = q₀; r ∈ nHat δ (q₀, w)⟧ ⟹ r ∈ F
 2. ⋀r p. ⟦r ∈ F; p = q₀; r ∈ nHat δ (q₀, w)⟧
          ⟹ r ∈ nHat δ (q₀, w)
*)
apply assumption
apply assumption
done;


(* Having proved both directions, we can show that L(M) = L(M') *)

lemma
  fixes M :: "('Q, 'Σ) nfa" and M' :: "('Q set, 'Σ) dfa"
  assumes M: "M = (q₀, δ, F)"
  assumes M': "M' = ({q₀}, δ', F')"
  assumes δ': "∀R a. δ'(R, a) = (⋃r∈R. δ(r, a))"
  assumes F': "F' = {R. ∃r∈R. r ∈ F}"
  shows "nL(M) = dL(M')"
apply (rule equalityI)
(*
goal (2 subgoals):
 1. nL M ⊆ dL M'
 2. dL M' ⊆ nL M
*)
apply (rule subsetI)
(*
goal (2 subgoals):
 1. ⋀x. x ∈ nL M ⟹ x ∈ dL M'
 2. dL M' ⊆ nL M
*)
apply (erule nL_dL [OF assms, rule_format])
(*
goal (1 subgoal):
 1. dL M' ⊆ nL M
*)
apply (rule subsetI)
(*
goal (1 subgoal):
 1. ⋀x. x ∈ dL M' ⟹ x ∈ nL M
*)
apply (erule dL_nL [OF assms, rule_format])
done;



end
```