Institute of Mathematics for Industry,
Kyushu University

Logic and Computation Project

Coq Modules for Relational Calculus

(Ver.0.1)

Hisaharu Tanaka
Saga University

Toshiaki Matsushima
Kyushu University

Shuichi Inokuchi
Fukuoka Institute of Technology

Yoshihiro Mizoguchi
Kyushu University

June 13, 2025

Repository: `https://github.com/KyushuUniversityMathematics/RelationalCalculus`

Contents

Chapter 1

Library Automaton

```
From mathcomp Require Import fintype finset seq.
Require Import MyLib.RelationalCalculus.

Module main(def:Relation).
Module Basic_Lemmas := Basic_Lemmas.main(Rel_Set).
Module Relation_Properties := Relation_Properties.main(Rel_Set).
Module Sum := Sum_Product.main Rel_Set.
Module Tac := Tactics.main Rel_Set.
Module Pta := Point_Axiom.main Rel_Set.

Import Rel_Set Basic_Lemmas Relation_Properties Sum Tac Pta.
Declare Scope automaton_scope.
Delimit Scope automaton_scope with AUTO.
Declare Scope language_scope.
Delimit Scope automaton_scope with LANG.
```

ああ s m f f j p ∪ ∩ ⊆

```
Structure automaton{state symbol:finType} :=
  {delta:symbol → (Rel state state); init:Rel i state; final:Rel state i}.

Fixpoint dstar {state symbol:finType}(d:symbol → (Rel state state))(w:seq symbol):(Rel
state state):=
match w with
|nil ⇒ Id state
|s::w' ⇒ (d s) · (dstar d w')
end.
Definition accept{state symbol:finType}(M:@automaton state symbol)(w:seq symbol):Prop
:=
(init M) · (dstar(delta M)w) · (final M) = Id i.

Open Scope language_scope.
Definition language {symbol:finType}:=seq symbol→Prop.
Definition rev_l {symbol:finType}(l:language):language:=
```

```
fun s:seq symbol⇒l (rev s).
Notation "l '^r'" := (rev_l l) (at level 30):language_scope.
Definition preimage_lang_l {symbol:finType}(l l':language):language:=
fun v:seq symbol⇒∃ u, l u ∧ l' (u++v).
Definition preimage_lang_r {symbol:finType}(l l':language):language:=
fun v:seq symbol⇒∃ u, l u ∧ l' (v++u).

Definition cup_lang{symbol:finType}(x y:language):language:=
fun s:seq symbol⇒x s∨y s.
Notation "x '∪' y" := (cup_lang x y)(at level 50):language_scope.
Definition cap_lang{symbol:finType}(x y:language):language:=
fun s:seq symbol⇒x s∧y s.
Notation "x '∩' y" := (cap_lang x y) (at level 50):language_scope.
Definition prod_lang{symbol:finType}(x y:language):language:=
fun s:seq symbol⇒∃ a b,s=a++b∧x a∧y b.
Notation "x '•' y" := (prod_lang x y) (at level 50):language_scope.
Definition phi{symbol:finType}:@language symbol := fun x⇒False.
Definition eps{symbol:finType}:language := fun s:seq symbol⇒s = nil.
Definition singl{symbol:finType}(s:symbol):language := fun x⇒ x = [::s].
Fixpoint power_l {symbol : finType}(n : nat)(l : @language symbol): language :=
  match n with
  | 0 ⇒ eps
  | S n' ⇒ l • power_l n' l
  end.
Notation "l '^' n" := (power_l n l) (at level 30):language_scope.
Definition star_l{symbol:finType}(l:language):language:=
fun s:seq symbol⇒ ∃ n,power_l n l s.
Notation "l '^*'" := (star_l l) (at level 30):language_scope.
Definition plus_l{symbol:finType}(l:language):language:=
fun s:seq symbol⇒ ∃ n,power_l(S n)l s.
Notation "l '^+'" := (plus_l l) (at level 30):language_scope.
Definition comp_lang{symbol:finType}(l:language):language:=
fun s:seq symbol=>~ l s.
Notation "l '^c'" := (comp_lang l) (at level 30):language_scope.

Fixpoint is_shuffle {A:Type} (u v w : seq A) : Prop :=
  match u, v, w with
  | nil, _, _ ⇒ v = w
  | _, nil, _ ⇒ u = w
  | a::u', b::v', c::w' ⇒ (a = c ∧ is_shuffle u' v w')\/(b = c ∧ is_shuffle u v' w')
  | _, _, _ ⇒ False
  end.
Compute is_shuffle[::1;2;3][::4;5;5][::1;2;4;3;5;5].
Definition shuffle_lang {symbol:finType}(l l':@language symbol):language :=
```

```
fun x⇒ ∃ u v, l u ∧ l' v ∧ is_shuffle u v x.

Definition deterministic {state symbol:finType}(M:@automaton state symbol):Prop:=
(∀ s:symbol,function_r(delta M s))/\function_r(init M).

Definition is_regular {symbol:finType}(l:language):Prop:=
∃ (state:finType)(M:@automaton state symbol),∀ w,l w ↔ accept M w.
Close Scope language_scope.

Definition plusM {state state' symbol:finType}(M:@automaton state symbol)(M':@automaton
state' symbol):automaton :=
     {|delta := fun x⇒Rel_sum(delta M x • inl_r _ state')(delta M' x • inr_r state _);
       init := Rel_sum(init M #)(init M' #)#;
       final := Rel_sum(final M)(final M')|}.
Notation "M '+' M'" := (plusM M M') (at level 50, left associativity):automaton_scope.
Definition timesM {state state' symbol:finType}(M:@automaton state symbol)(M':@automaton
state' symbol):automaton :=
     {|delta := fun s⇒Rel_prod(fst_r state state' • delta M s)(snd_r state state' • delta
M' s);
       init := Rel_prod(init M)(init M');
       final := Rel_prod(final M #)(final M' #)# |}.
Notation "M '×' M'" := (timesM M M') (at level 50, left associativity):automaton_scope.
Definition revM {state symbol:finType}(M:@automaton state symbol):=
     {|delta := fun x⇒delta M x #; init := final M #; final := init M # |}.
Definition concatM {state state' symbol:finType}(M:@automaton state symbol)(M':@automaton
state' symbol):automaton:=
     {|delta := fun x =>( @inl_r _ state' #•delta M x•@inl_r _ state') ∪ (Rel_sum(final
M • init M')(Id _) • delta M' x • @inr_r state _);
       init := init M • (Rel_sum(Id _)(init M' # • final M #)#);
       final := (Rel_sum(final M • init M')(Id _)) • final M'|}.
Definition phiM{symbol:finType}:@automaton _ symbol:= {|delta := fun x=>φ i i; init
:= φ i i;final := φ i i|}.
Definition epsM{symbol:finType}:@automaton _ symbol:={|delta := fun x=>φ i i;init:=Id
i;final:=Id i|}.
Definition singlM{symbol:finType}(s:symbol):automaton :=
     {|delta := fun (x:symbol)(a b:bool)=>a∧x=s∧not b;
       init := fun (x:i)(a:bool)=>is_true a;
       final := fun (a:bool)(x:i)=>not (is_true a)|}.
Definition compM{state symbol:finType}(M:@automaton state symbol):automaton:=
     {|delta := delta M; init := init M; final := final M ˆ|}.
Definition plus_nfa {state symbol:finType}(M:@automaton state symbol):automaton :=
     {|delta := fun x=>(delta M x ∪ (delta M x • (final M • init M)));
       init := init M;
       final := final M|}.
Definition shuffle_nfa {state state' symbol:finType}
```

```
  (M:@automaton state symbol)(M':@automaton state' symbol):automaton :=
  {|delta := fun x⇒Rel_prod(fst_r state state'•delta M x)(snd_r state state') ∪ Rel_prod(fst_r
state state')(snd_r state state' • delta M' x);
    init := Rel_prod(init M)(init M');
    final := Rel_prod(final M #)(final M' #)# |}.
Definition preimage_nfa {state state' symbol:finType}
  (M:@automaton state symbol)(M':@automaton state' symbol):=
  {|delta := delta M';
    init := init M' • fun x y⇒∃ w,accept M w/\(dstar(delta M')w)x y;
    final := final M'|}.

Ltac destruct_Id_i :=
  repeat match goal with
  | [H : _ = Id _ ⊢ _ ] ⇒
    have:Id i tt tt by [];rewrite-{1}H=>{}H
  | [_ : _ ⊢ _ = Id _] ⇒
    apply/inc_antisym_eq;split;
    [by rewrite unit_identity_is_universal|move=>[][] _]
  end.

Lemma cup_idi:∀ alpha beta,
  alpha ∪ beta = Id i↔alpha = Id i∨beta = Id i.
Proof.
move⇒alpha beta.
case:( @unit_empty_or_universal beta)=>H;rewrite H;
[rewrite cup_empty|rewrite cup_universal].
move:unit_identity_not_empty=>{}H.
by split=>[|[|H0]];[left| |rewrite H0 in H].
rewrite unit_identity_is_universal.
by split=>[|[]];[right| |].
Qed.

Lemma cap_idi:∀ alpha beta,
  alpha ∩ beta = Id i↔alpha = Id i∧beta = Id i.
Proof.
move⇒alpha beta.
case:( @unit_empty_or_universal beta)=>H;rewrite H;
[rewrite cap_empty|rewrite cap_universal].
move:unit_identity_not_empty=>{}H.
split=>[H0|[] _ H0];by rewrite H0 in H.
rewrite unit_identity_is_universal.
by split=>[|[]].
Qed.

Lemma dstar_cat {state symbol:finType}:
```

```
∀(d:symbol→Rel state state)(a b:seq symbol),
dstar d(a++b) = dstar d a • dstar d b.
Proof.
move⇒d a b.
elim:a=>[|h a H].
by rewrite/=comp_id_l.
by rewrite/=H comp_assoc.
Qed.

Theorem NFA_DFA_equiv {state symbol:finType}(M:@automaton state symbol):
∃(state':finType)(M':@automaton state' _),deterministic M'∧
(∀ w,accept M w ↔ accept M' w).
Proof.

destruct M.
∃ _,{|delta:= fun (s:symbol)(x y:{set state})=>[set z | is_true_inv(exists2 w, w \in x
& delta0 s w z)] = y;
          init:= fun (x:i)(y:{set state})=>[set x|is_true_inv(init0 tt x)]=y;
          final:= fun(y:{set state})(x:i)=>y:&:[set x|is_true_inv(final0 x tt)]<>set0|}.

rewrite/deterministic/accept/=.
split.

split=>[s|];rewrite/=/function_r/total_r/univalent_r;split.
move⇒x y H.
have{}H:y=x;[done|subst].
by ∃ [set z | is_true_inv (exists2 w0 : state, w0 \in x & delta0 s w0 z)].
case⇒alpha x[]y[]{}H H1.
rewrite/inverse in H.
by subst.
move=>[][] _.
by ∃ [set x | is_true_inv (init0 tt x)].
case⇒alpha x[]y[]{}H H1.
rewrite/inverse in H.
by subst.

move⇒w.
move:init0.
elim:w=>[|h w H]init0.
rewrite/=!comp_id_r.
split⇒H;destruct_Id_i.
case:H⇒a[]H H0.
∃ [set x | is_true_inv (init0 tt x)].
split;[done|]=>H1.
move:(in_set0 a).
rewrite-{}H1 in_setI!in_set=>/andP[].
```

```
by split;apply/is_true_id.
case:H⇒a'[]H/eqP/set0Pn H0.
subst.
case:H0⇒a.
rewrite in_setI!in_set=>/andP[]/is_true_id H/is_true_id H'.
by ∃ a.

rewrite/=-!comp_assoc{}H.
have H:∀ x,(exists2 w0 : state,
w0 \in [set x0 | is_true_inv (init0 tt x0)] & delta0 h w0 x)<->
(init0 • delta0 h) tt x.
move⇒x.
split=>[][]y.
rewrite in_set=>/is_true_id H H0.
by ∃ y.
case=>/is_true_id H H0.
by ∃ y;[rewrite in_set|].
have{}H:∀ x : state,
(exists2 w0 : state,
w0 \in [set x0 | is_true_inv (init0 tt x0)] &
delta0 h w0 x) = (init0 • delta0 h) tt x
by move⇒x;apply/prop_extensionality_ok.

split⇒H0;destruct_Id_i.
case:H0⇒a[][]b[]H0 H1 H2.
∃ a.
split;[move{H2};∃ b|done].
split;[|done].
∃ [set x | is_true_inv (init0 tt x)].
split;[done|].
rewrite-{H1}H0.
apply/setP⇒x.
by rewrite!in_set H.

rewrite!comp_assoc-comp_assoc in H0 ×.
case:H0⇒a[]H0 H1.
∃ a.
split;[move{H1}|done].
case:H0⇒b[]H0 H1.
rewrite-{}H1-{}H0.
apply/setP⇒x.
by rewrite!in_set H.
Qed.

Open Scope language_scope.
```

```
Open Scope automaton_scope.
Theorem regular_cup {symbol:finType}(l l':@language symbol):
is_regular l∧is_regular l'→is_regular (l ∪ l').
Proof.
Close Scope language_scope.
rewrite/is_regular=>[][][][]state[]M H[]state'[]M' H'.
∃ (sum state state'),(M + M') ⇒ w.
rewrite/cup_lang{}H{}H'/accept.

destruct M,M';simpl.
remember(fun x : symbol ⇒Rel_sum (delta0 x • inl_r state state')
(delta1 x • inr_r state state')) as d.
rewrite-Heqd.
move:init0 init1.
elim:w=>[|h w H]init0 init1.
by rewrite/=!comp_id_r sum_comp 2!inv_invol cup_idi.

have H0:Rel_sum (init0 #) (init1 #) # • d h =
Rel_sum ((init0 • delta0 h) #) ((init1 • delta1 h) #) #.
by rewrite Heqd sum_comp/Rel_sum inv_cup_distr!comp_inv 6!inv_invol!comp_assoc.
rewrite/=-!comp_assoc{}H!comp_assoc-(comp_assoc _ _ _ _ _(d h)).
by split⇒H;rewrite-H;f_equal.
Qed.

Open Scope language_scope.
Theorem regular_cap {symbol:finType}(l l':@language symbol):
is_regular l∧is_regular l'→is_regular (l ∩ l').
Proof.
Close Scope language_scope.
rewrite/is_regular=>[][][][]state[]M H[]state'[]M' H'.
∃ (prod state state'),(M × M')=>w.
rewrite/cap_lang{}H{}H'/accept.
remember(delta (M × M'))as d.
destruct M,M'.
rewrite/= in Heqd ×.
rewrite-Heqd.

move:init0 init1.
elim:w=>[|h w H]init0 init1.
by rewrite/=!comp_id_r/Rel_prod inv_cap_distr!comp_inv-sharpness
2!inv_invol cap_idi.
rewrite/=-!comp_assoc{}H!comp_assoc-(comp_assoc _ _ _ _ _(d h)).
have H:Rel_prod init0 init1 • d h =
Rel_prod (init0 • delta0 h) (init1 • delta1 h).
by rewrite Heqd/Rel_prod!comp_assoc
```

```
-(inv_invol _ _(delta0 h · fst_r state state' #))
-(inv_invol _ _(delta1 h · snd_r state state' #))-sharpness.
by split⇒H0;rewrite-H0;f_equal.
Qed.

Open Scope language_scope.
Theorem regular_rev{symbol:finType}(l:@language symbol):
is_regular l → is_regular (l^r).
Proof.
Close Scope language_scope.
rewrite/is_regular=>[][]state[]M H.
∃ state,(revM M)=>w.
rewrite/rev_l{l}H/accept/revM/=.
have H:∀ (d:symbol→Rel state state)(w:seq symbol)(h:symbol),dstar d (rcons w h)=dstar
d w · d h.
move⇒d{}w h.
elim:w;[by rewrite/=comp_id_l comp_id_r|]=>a w H.
by rewrite/=comp_assoc H.
have{}H:dstar (fun x : symbol ⇒ delta M x #) w = dstar (delta M) (rev w) #.
elim:w;[by rewrite inv_id|]=>h w H0.
by rewrite/={}H0 rev_cons H comp_inv.
by split⇒H0;rewrite-inv_id-{}H0!comp_inv inv_invol comp_assoc H.
Qed.

Open Scope language_scope.
Theorem regular_prod{symbol:finType}(l l':@language symbol):
is_regular l ∧ is_regular l' → is_regular(l · l').
Proof.
Close Scope language_scope.
rewrite/is_regular/accept/prod_lang=>[][][]state[]M H[]state'[]M' H'.
∃ (sum state state'),(concatM M M')=>w.
remember (delta (concatM M M')) as d.
remember (final (concatM M M')) as f.
destruct M,M'.
rewrite/= in H H' Heqd Heqf×.

have{H'}H:(∃ u v : seq symbol, w = u ++ v ∧ l u ∧ l' v)<->
(∃ u v : seq symbol, w = u ++ v ∧
(init0 · dstar delta0 u) · final0 ·(init1 · dstar delta1 v) · final1 = Id i).
split=>[[]u[]v[]H0[]/H{}H/H'{}H'|[]u[]v[]H0 H1];∃ u,v.
by rewrite H comp_id_l.
case:( @unit_empty_or_universal((init0 · dstar delta0 u) · final0))=>H2;
rewrite H2 in H1 ×.
move:unit_identity_not_empty.
by rewrite-H1!comp_empty_l.
```

```
by rewrite H H' H2-{2}H1-unit_identity_is_universal comp_id_l.
rewrite{}H-Heqd-Heqf.
have inil:∀ init:Rel i state,init · Rel_sum (Id state) (init1 # · final0 #) # · inl_r state
state'# = init.
move⇒init.
by rewrite comp_assoc/Rel_sum comp_id_r inv_cup_distr!comp_inv
2!inv_invol(inv_invol _ _(inr_r _ _))comp_cup_distr_r!comp_assoc
inl_id inr_inl_empty!comp_empty_r cup_empty comp_id_r.
have rf:inr_r state state' · f = final1.
rewrite Heqf-comp_assoc-{2}( @comp_id_l _ _ final1).
f_equal.
by rewrite/Rel_sum comp_cup_distr_l-!comp_assoc inr_inl_empty inr_id
!comp_empty_l cup_comm comp_id_l cup_empty.
have lf:inl_r state state' · f = final0 · (init1 · final1).
rewrite Heqf-!comp_assoc.
f_equal.
by rewrite/Rel_sum comp_cup_distr_l-!comp_assoc inl_inr_empty inl_id
comp_empty_l comp_id_l cup_empty.
have {Heqf}inif:∀ init:Rel i state,
init · Rel_sum (Id state) (init1 # · final0 #) # · f = init · final0 · init1 · final1.
move⇒init.
rewrite Heqf!comp_assoc.
f_equal.
rewrite-!comp_assoc.
f_equal.
by rewrite sum_comp comp_inv 2!inv_invol inv_id comp_id_l comp_id_r cup_idem.
have rdstar:∀ w,inr_r state state' · dstar d w = dstar delta1 w · inr_r state state'.
elim=>[|h{}w H].
by rewrite/=comp_id_l comp_id_r.
by rewrite/=-comp_assoc{1}Heqd/Rel_sum!comp_cup_distr_r!comp_cup_distr_l
-!comp_assoc inr_id inr_inl_empty!comp_empty_l!comp_id_l cup_comm
cup_empty cup_comm cup_empty!comp_assoc H.
have{rf rdstar}rdstarf:∀ w,inr_r state state' · (dstar d w · f) = dstar delta1 w · final1.
move=>{}w.
by rewrite-comp_assoc rdstar comp_assoc rf.
have dstarl:∀ w,dstar d w · inl_r state state'# =
inl_r state state'# · dstar delta0 w.
elim=>[|h{}w H].
by rewrite/=comp_id_l comp_id_r.
rewrite/=comp_assoc{}H-!comp_assoc.
f_equal.
```

```
by rewrite Heqd/Rel_sum!comp_cup_distr_r!comp_assoc inr_inl_empty inl_id
comp_id_r comp_id_l!comp_empty_r!cup_empty.

have {inil}inidstarl:∀ (init:Rel i state)(w:seq symbol),init · Rel_sum (Id state) (init1 # ·
final0 #) # · dstar d w · inl_r state state'# =init · dstar delta0 w.
move⇒init{}w.
by rewrite comp_assoc dstarl-comp_assoc inil.

have{}Heqd:∀ h,d h = inl_r state state'# · delta0 h · inl_r state state' ∪
(inl_r state state'# · final0 · init1 · delta1 h · inr_r state state') ∪
(inr_r state state'# · delta1 h · inr_r state state').
move⇒h.
by rewrite Heqd/Rel_sum comp_id_r cup_assoc!comp_cup_distr_r!comp_assoc.

split=>[[]u[]v[]H H0|H].
rewrite{w}H dstar_cat.

rewrite-(comp_id_r _ _(dstar d u))-inl_inr_cup_id comp_cup_distr_l
!comp_cup_distr_r comp_cup_distr_l comp_cup_distr_r-!comp_assoc inidstarl.
suff H2:((((init0 · dstar delta0 u) · inl_r state state') · dstar d v) · f)=Id i.
by rewrite H2 unit_identity_is_universal cup_comm cup_universal.

move:H0.
case:v=>[|h v].
rewrite/=!comp_id_r⇒H.
by rewrite!comp_assoc lf-!comp_assoc H.

rewrite/=-!comp_assoc⇒H.
by rewrite Heqd!comp_cup_distr_l!comp_cup_distr_r-!comp_assoc
!(comp_assoc _ _ _ _ _(inl_r _ _)) inl_id inl_inr_empty comp_empty_r
!comp_empty_l cup_empty comp_id_r!comp_assoc rdstarf-!comp_assoc H
unit_identity_is_universal cup_universal.

rewrite/Rel_sum comp_id_r inv_cup_distr!comp_inv!inv_invol
comp_cup_distr_l!comp_cup_distr_r cup_idi in H.
case:H.

move:init0.
elim:w=>[|h w H]init0.
rewrite/=comp_id_r comp_assoc lf⇒H.
∃ nil,nil.
by rewrite/=!comp_id_r!comp_assoc H.
rewrite/=-comp_assoc{1}Heqd!comp_cup_distr_l!comp_cup_distr_r
-!comp_assoc!(comp_assoc _ _ _ _ _(inl_r _ _))inl_id comp_id_r
inl_inr_empty comp_empty_r!comp_empty_l cup_empty cup_idi-!comp_assoc
=>[[[/H[]u[]v[]{}H H0|{}H].
∃(h::u),v.
rewrite/=-!comp_assoc in H0 ×.
```

```
by split;[f_equal||].
∃ nil,(h::w).
by rewrite/=comp_id_r!comp_assoc rdstarf in H ×.
move⇒H.
∃ nil,w.
by rewrite/=comp_id_r!comp_assoc rdstarf in H ×.
Qed.

Theorem regular_phi{symbol:finType}:@is_regular symbol phi.
Proof.
rewrite/is_regular.
∃ i,phiM⇒w.
by split;[|rewrite/accept/=!comp_empty_l⇒H;move:unit_identity_not_empty;rewrite H].
Qed.
Theorem regular_eps{symbol:finType}:@is_regular symbol eps.
rewrite/is_regular.
∃ i,epsM.
rewrite/accept.
case=>[|a l].
by split=>[_|];[rewrite/=!comp_id_l|].
rewrite/=comp_empty_l!comp_empty_r comp_empty_l.
by split=>[|H];[|move:unit_identity_not_empty;rewrite H].
Qed.

Theorem regular_singl{symbol:finType}(s:symbol):is_regular(singl s).
Proof.
rewrite/is_regular.
∃ bool,(singlM s).
have H:∀ w,
  (dstar(delta(singlM s))w)true false↔accept(singlM s) w⇒w.
rewrite/accept.
split⇒H;destruct_Id_i.
∃ false.
by split;[∃ true|].
by case:H=>[][][];[by simpl|]=>[][][][].
rewrite-{}H.

case:w=>[|a[|b w]];[done|rewrite/=comp_id_r|simpl].
split=>[[]|[] _[]H _];by subst.
by split=>[[]|[]x[][] _[] _ H[]y[][]].
Qed.

Open Scope language_scope.
Theorem regular_comp{symbol:finType}(l:@language symbol):
is_regular l → is_regular (l^c).
```

```
Proof.
Close Scope language_scope.
rewrite/is_regular/comp_lang=>[][]state[]M H.
move:(NFA_DFA_equiv M)=>[]state'[]M'[]H0 H1.
∃ _,(compM M')=>w.
rewrite{l}H{state M}H1.

rewrite/accept/=.
move:H0.
rewrite/deterministic=>[][]H H0.
have{H0}H:function_r(init M' • dstar(delta M')w).
destruct M';rewrite/= in H H0 ×.
move:init0 H0.
elim:w=>[|h w H1]init0 H0.
by rewrite/=comp_id_r.
rewrite/=-comp_assoc.
apply/H1/function_comp/H/H0.

have H0:∃ x, (init M' • dstar(delta M')w)tt x.
rewrite/function_r/total_r in H.
move:H=>[]{}H _.
have H0:Id i tt tt;[done|].
move:(H tt tt H0)=>[]x[]{H0}H _.
by ∃ x.
case:H0⇒x H0.
have{}H:∀ y,(init M' • dstar (delta M') w) tt y → y=x.
move⇒y H1.
rewrite/function_r/univalent_r in H.
move:H=>[] _ H.
have{H1}H0:((init M' • dstar (delta M') w) # • (init M' • dstar (delta M') w))x y.
by ∃ tt.
by move:(H _ _ H0).

have{H0}H:∀ R:Rel state' i,
(init M' • dstar (delta M') w) • R = Id i ↔ R x tt.
move⇒R.
split⇒H1;destruct_Id_i.
case:H1⇒y[]/H{}H.
by subst.
by ∃ x.
by rewrite/not!H/complement/rpc.
Qed.

Lemma nil_rcons {symbol:Type}(s:seq symbol):s = nil ∨∃ t s',s=rcons s' t.
Proof.
```

```
elim:s=>[|h s[H|[]h'[]s']];[by left| |];right;
[∃ h,nil|∃ h',(h::s')];by subst.
Qed.
Lemma dstar_rcons {state symbol:finType}:
∀(d:symbol→Rel state state) w t,
dstar d(rcons w t) = dstar d w • d t.
Proof.
move⇒d w t.
elim:w=>[|h w H].
by rewrite/=comp_id_l comp_id_r.
by rewrite/=H comp_assoc.
Qed.

Theorem plus_regular {symbol:finType}(l:@language symbol):
is_regular l → is_regular(plus_l l).
Proof.
rewrite/is_regular=>[][]state[]M H.
∃ _, (plus_nfa M)=>w.
remember(delta(plus_nfa M))as d.
destruct M.
rewrite/accept/plus_l/= in H Heqd ×.
rewrite-Heqd.

split.
case⇒n.

have H0:∀ w, ((init0 • dstar delta0 w) • final0)
⊆ ((init0 • dstar d w) • final0).
move=>{}w.
apply/comp_inc_compat_ab_a'b/comp_inc_compat_ab_ab'.
elim:w=>[|h{}w{}H];[done|simpl⇒x z[]y[]H0 H1].
∃ y.
rewrite{1}Heqd.
by split;[apply/cup_l|apply/H].

move:w.
elim:n=>[|n H1]w.
rewrite/=/prod_lang=>[][]a[][|b' b[] _[] _];[|done].
rewrite cats0=>[][]H1[]/H{}H _.
destruct_Id_i.
move:H=>/H0{H0}H.
by subst.

rewrite/={1}/prod_lang=>[][]a[]b[]H2[]/H{}H/H1{}H1.
rewrite{w}H2 dstar_cat.
move:H.
```

```
have H:(∀ x:seq symbol,x = nil ∨∃ h x', x = rcons x' h).
elim=>[|h x H];[by left|right].
case:H⇒H.
∃ h,nil.
by subst.
case:H⇒x0[]x1 H.
subst.
by ∃ x0,(h::x1).
case:(H a)=>[|[]t[]a'][]{}H.
by rewrite{a}H/=comp_id_l.
rewrite{a}H.
have dstar_rcons:∀ d,dstar d(rcons a' t) = dstar d a' • d t.
move⇒t0{Heqd H0 H1}d.
elim:a'=>[|h a H].
by rewrite/=comp_id_r comp_id_l.
by rewrite/=H comp_assoc.
rewrite!dstar_rcons⇒H.
rewrite{2}Heqd-!comp_assoc!comp_cup_distr_l!comp_cup_distr_r-!comp_assoc.
have{}H:init0 • dstar d a' • delta0 t • final0 = Id i.
apply/inc_antisym_eq.
split;[by rewrite unit_identity_is_universal|rewrite-{}H!comp_assoc].
apply/comp_inc_compat_ab_ab'/comp_inc_compat_ab_a'b.
elim:{dstar_rcons}a'=>[|h a H].
done.
rewrite/={1}Heqd comp_cup_distr_r comp_assoc-comp_cup_distr_l.
apply/comp_inc_compat_ab_ab'⇒x y/H.
apply/cup_l.
by rewrite H comp_id_l H1 unit_identity_is_universal cup_universal.
have{}H:∀ w,(init0 • dstar d w) • final0 = Id i →
l w∨∃ u v, w = u ++ v ∧ size v < size w ∧
  l u ∧ init0 • dstar d v • final0 = Id i.
move=>{}w H0.
destruct_Id_i.
case:H0=>[]b[][]a[]{}H0 H1 H2.
have:(dstar delta0 w • final0) a tt ∨ ∃ n,
(dstar delta0(take(S n)w) • final0) a tt /\(init0 • dstar d(drop(S n)w) • final0)tt tt.
move:a{H0}H1.
elim:w=>[|h w H0]a.
rewrite/=comp_id_l comp_id_r⇒H1.
have{}H1:a = b;[done|subst].
by left.
rewrite/={1}Heqd/cup/cupP=>[][][]c[][]alpha[][]H' H1;rewrite{alpha}H' in H1.
```

```
move/H0.
case⇒H3.
left.
rewrite comp_assoc.
by ∃ c.
case:H3⇒n[]H3 H4.
right.
∃(S n).
rewrite comp_assoc.
by split;[∃ c|].

move=>{}H0.
right.
∃ 0.
rewrite take0 drop0/=comp_id_r.
rewrite-comp_assoc in H1.
case:H1=>[][][]H1 H3.
split;[done|∃ b].
by split;[∃ c|done].

case⇒H3;[left|].
rewrite H comp_assoc.
destruct_Id_i.
by ∃ a.
case:H3⇒n[]H3 H4.
case_eq(w==nil)=>/eqP H5;[left|right].
rewrite H H5/=comp_id_r in H1 ×.
Rel_simpl;[by rewrite unit_identity_is_universal|move=>[][] _].
have{}H1:a=b;[done|subst].
by ∃ b.

∃(take (S n) w),(drop (S n) w).
rewrite cat_take_drop size_drop.
split;[done|split].
move:{H1 H3 H4}H5.
case:w=>[|h w];[done|simpl=>{h}_].
rewrite ssrnat.subnE PeanoNat.Nat.sub_succ.
apply/PeanoNat.le_lt_n_Sm/PeanoNat.Nat.le_sub_l.
split.
rewrite H.
destruct_Id_i.
rewrite comp_assoc.
by ∃ a.
by destruct_Id_i.
```

```
have:size w < S(size w)by [].
remember(S(size w)) as n=>{Heqn}.
move:w.
elim:n=>[|n H0]w.
by move/PeanoNat.Nat.nlt_0_r.
move⇒H1/H.
case⇒H2.
∃ 0,w,nil.
by rewrite cats0.
case:H2⇒u[]v[]H2[]H3[]H4.
have{H3 H1}:size v < n.
apply/PeanoNat.Nat.lt_le_trans/PeanoNat.lt_n_Sm_le/H1/H3.
move/H0=>{}H0/H0[]n'{}H.
by ∃ (S n'),u,v.
Qed.

Open Scope language_scope.
Theorem star_regular {symbol:finType}(l:@language symbol):
is_regular l → is_regular (l^*).
Proof.
move/plus_regular⇒H.
move:( @regular_eps symbol)=>H0.
have{H0}H:is_regular (plus_l l ∪ eps)by apply/regular_cup.
rewrite/star_l/is_regular/plus_l in H ×.
case:H⇒state[]M H.
∃ _,M⇒w.
rewrite-{state M}H/cup_lang.
split.
by case=>[][|n]H;[right|left;∃ n].
by case=>[[]n H|];[∃ (S n)|∃ 0].
Qed.

Close Scope language_scope.

Theorem shuffle_regular {symbol:finType}(l l':@language symbol):
is_regular l ∧ is_regular l' → is_regular(shuffle_lang l l').
Proof.
rewrite/is_regular=>[][][]state[]M H[]state'[]M' H'.
∃ _,(shuffle_nfa M M')=>w.
remember (delta(shuffle_nfa M M'))as d.
remember (final(shuffle_nfa M M'))as f.
destruct M,M'.
rewrite/accept/= in H H' Heqd Heqf ×.
rewrite-Heqd-Heqf/shuffle_lang.
```

```
have d_inv:∀ x,d x =
  Rel_prod(fst_r state state' • delta0 x #)(snd_r state state')#∪
  Rel_prod(fst_r state state')(snd_r state state' • delta1 x #)#.
move⇒x.
rewrite Heqd.
by f_equal;rewrite/Rel_prod inv_cap_distr!comp_inv 3!inv_invol!comp_assoc.

have sharpness':∀ alpha beta gamma delta,
Rel_prod alpha beta •
Rel_prod(fst_r state state' • gamma #)(snd_r state state' • delta #)# =
Rel_prod(alpha • gamma)(beta • delta).
move⇒t t0 t1 alpha beta gamma delta.
rewrite/Rel_prod inv_cap_distr.
remember(fst_r state state'  •  gamma #)as fg.
remember(snd_r state state'  •  delta #)as sd.
by rewrite!comp_inv 2!inv_invol-sharpness Heqfg Heqsd!comp_inv
2!inv_invol!comp_assoc.

have inid:∀ x(alpha:Rel i _)beta,Rel_prod alpha beta • d x =
  Rel_prod(alpha • delta0 x)beta ∪ Rel_prod alpha(beta • delta1 x).
move⇒x alpha beta.
by rewrite d_inv comp_cup_distr_l-{1}(comp_id_r _ _(snd_r _ _))
-{2}(comp_id_r _ _(fst_r _ _))-inv_id-(@inv_id state)!sharpness'
!comp_id_r.

have inif:∀(alpha:Rel i _)beta,Rel_prod alpha beta • f=alpha • final0 ∩ (beta • final1).
move⇒alpha beta.
by rewrite Heqf/Rel_prod inv_cap_distr!comp_inv
!(inv_invol (prod state state'))-sharpness 2!inv_invol.

have nilw:∀(init0:Rel i state)(init1:Rel i state')w,
  init0 • final0 = Id i∧init1 • dstar delta1 w • final1 = Id i
  → Rel_prod init0 init1 • dstar d w • f = Id i.
move=>{H}init0{H'}init1{}w[]{}H{}H'.
move:init1 H'.
elim:w=>[|h w H0]init1.
rewrite/=!comp_id_r⇒H'.
by rewrite inif H H' cap_idem.
rewrite/=-!comp_assoc inid !comp_cup_distr_r=>/H0{}H0.
by rewrite H0 unit_identity_is_universal cup_universal.

have wnil:∀(init0:Rel i state)(init1:Rel i state')w,
  init0 • dstar delta0 w • final0 = Id i∧init1 • final1 = Id i
  → Rel_prod init0 init1 • dstar d w • f = Id i.
move=>{H}init0{H'}init1{}w[]{}H{}H'.
move:init0 H.
```

```
elim:w=>[|h w H0]init0.
rewrite/=!comp_id_r⇒H.
by rewrite inif H H' cap_idem.
rewrite/=-!comp_assoc inid !comp_cup_distr_r=>/H0{}H0.
by rewrite H0 unit_identity_is_universal cup_comm cup_universal.

split=>[[]u[]v[]/H{}H[]/H'{}H'|].
move:init0 init1 u v H H'.
elim:w=>[|h w H0]init0 init1 u v H H'.
destruct u,v;[|done|done|done].
rewrite/=!comp_id_r in H H' ×.
by rewrite inif H H' cap_idem.
destruct u⇒H1.
have{}H1:v = h::w by destruct v,w.
rewrite-{w h H0}H1/=comp_id_r in H ×.
by apply/nilw.
destruct v.
remember(h::w)as w'.
have{}H0:s::u = w' by destruct(s::u),w'.
rewrite{s u H1}H0/=comp_id_r in H H'.
by apply/wnil.
rewrite/= in H1.
case:H1=>[][]H1 H2;[rewrite{s}H1 in H H0|rewrite{s0}H1 in H' H0];
[remember(s0::v)as v'|remember(s::u)as u'];
rewrite/=-!comp_assoc inid in H H' *;
move:(H0 _ _ _ _ H H' H2)=>{}H0;
rewrite!comp_cup_distr_r H0 unit_identity_is_universal;
[by rewrite cup_comm cup_universal|by rewrite cup_universal].

move⇒H0.
suff:∃ u v : seq symbol,
  init0 · dstar delta0 u · final0 = Id i ∧
  init1 · dstar delta1 v · final1 = Id i ∧ is_shuffle u v w.
move=>[]u[]v[]/H{}H[]/H'{}H' H1.
by ∃ u,v.
move:init0 init1{H H'}H0.
elim:w=>[|h w H0]init0 init1.
rewrite/=comp_id_r inif=>/cap_idi[]H0 H1.
∃ nil,nil.
by rewrite/=!comp_id_r.

rewrite/=-comp_assoc inid!comp_cup_distr_r=>/cup_idi[]/H0[]u[]v[]H[]{}H0 H1.
∃(h::u),v.
rewrite/=-comp_assoc.
split;[done|split;[done|]].
```

```
destruct v.
f_equal.
by destruct u,w.
by left.
∃ u,(h::v).
rewrite/=-comp_assoc.
split;[done|split;[done||]].
destruct u.
f_equal.
by destruct v,w.
by right.
Qed.
Theorem preimage_regular_l {symbol:finType}(l l':@language symbol):
  is_regular l → is_regular l' → is_regular(preimage_lang_l l l').
Proof.
rewrite/is_regular=>[][]state[]M H[]state'[]M' H'.
∃ _, (preimage_nfa M M')=>w.
destruct M,M'.
rewrite/accept/=/accept/preimage_lang_l/= in H H' ×.
split=>[[]u[]/H{}H/H'{}H'|H0].
rewrite dstar_cat!comp_assoc in H' ×.
destruct_Id_i.
case:H'⇒x[]H0[]y[]H1 H2.
∃ x.
split;[done|∃ y].
split;[∃ u|done].
by split;[destruct_Id_i|].
destruct_Id_i.
rewrite comp_assoc in H0.
case:H0⇒x[][]y[]H0[]u[]H1 H2 H3.
∃ u.
rewrite{}H{}H'{}H1 dstar_cat-comp_assoc comp_assoc.
split;[done|destruct_Id_i].
∃ x.
by split;[∃ y||].
Qed.
Lemma rev_invol {symbol:Type}:∀ l:seq symbol,rev(rev l) = l.
Proof.
by elim=>[|h w H];[|rewrite rev_cons rev_rcons H].
Qed.
Open Scope language_scope.
```

```
Theorem preimage_regular_r {symbol:finType}(l l':@language symbol):
  is_regular l∧is_regular l' → is_regular(preimage_lang_r l l').
Proof.
move=>[]/regular_rev H/regular_rev H'.
move:(preimage_regular_l _ _ H H')=>/regular_rev{H' H}.

rewrite/is_regular=>[][]state[]M H.
∃ _, M⇒w.
rewrite-{state M}H.
rewrite/preimage_lang_l/preimage_lang_r/rev_l.
split=>[][]u[]H0 H0'.
∃ (rev u).
by rewrite rev_cat!rev_invol.
∃ (rev u).
by rewrite rev_cat rev_invol in H0'.
Qed.

End main.
```

Bibliography

[1] R. Affeldt and M. Hagiwara. Formalization of Shannon 's Theorems in SSReflect-Coq. In 3rd Conference on Interactive Theorem Proving, LNCS 7406, 233–249, 2012.