CoqSticker Module (Ver.0.1)

早川 銀河
(九州大学大学院数理学府)
Email: `hayakawa.ginga.875@s.kyushu-u.ac.jp`

溝口 佳寛
(九州大学マス・フォア・インダストリ研究所)

2024 年 9 月 25 日

目 次

第1章　Library AutomatonEx

```
From mathcomp Require Import all_ssreflect.
Require Import AutomatonModule.

Inductive Z2 := zero|one.
Inductive ab := a|b.

Definition Z2_eqb(x1 x2:Z2) :=
match x1,x2 with |zero,zero⇒true|one,one⇒true|_,_⇒false end.
Definition ab_eqb(x1 x2:ab) :=
match x1,x2 with |a,a⇒true|b,b⇒true|_,_⇒false end.

Lemma eq_Z2P:Equality.axiom Z2_eqb.
Proof. move⇒x y;apply: (iffP idP); rewrite /eq_ascii; by destruct x,y.
Qed.
Lemma eq_abP:Equality.axiom ab_eqb.
Proof. move⇒x y;apply: (iffP idP); rewrite /eq_ascii; by destruct x,y.
Qed.

Definition Z2_eqMixin := EqMixin eq_Z2P.
Canonical Z2_eqType := Eval hnf in EqType _ Z2_eqMixin.
Definition ab_eqMixin := EqMixin eq_abP.
Canonical ab_eqType := Eval hnf in EqType _ ab_eqMixin.

Compute zero==one.

Definition nat_of_Z2(x:Z2):=match x with zero=¿0—one=¿1 end.
Definition Z2_of_nat(n:nat):=match n with 0=¿Some zero—1=¿Some
one|_⇒None end.
Definition nat_of_ab(x:ab):=match x with a=¿0—b=¿1 end.
Definition ab_of_nat(n:nat):=match n with 0=¿Some a—1=¿Some b|_⇒None
end.

Lemma Z2_count_spec :pcancel nat_of_Z2 Z2_of_nat.
Proof. rewrite/pcancel⇒x;by destruct x. Qed.
Lemma ab_count_spec :pcancel nat_of_ab ab_of_nat.
Proof. rewrite/pcancel⇒x;by destruct x. Qed.

Definition Z2_countMixin := CountMixin Z2_count_spec.
Definition Z2_choiceMixin := CountChoiceMixin Z2_countMixin.
```

```
Canonical Z2_choiceType := Eval hnf in ChoiceType Z2 Z2_choiceMixin.
Canonical Z2_countType := Eval hnf in CountType Z2 Z2_countMixin.
Definition ab_countMixin := CountMixin ab_count_spec.
Definition ab_choiceMixin := CountChoiceMixin ab_countMixin.
Canonical ab_choiceType := Eval hnf in ChoiceType ab ab_choiceMixin.
Canonical ab_countType := Eval hnf in CountType ab ab_countMixin.

Definition enum_Z2 := [::zero;one].
Definition enum_ab := [::a;b].

Lemma enum_Z2P :Finite.axiom enum_Z2.
Proof. rewrite/Finite.axiom⇒x;by destruct x. Qed.
Lemma enum_abP :Finite.axiom enum_ab.
Proof. rewrite/Finite.axiom⇒x;by destruct x. Qed.

Definition Z2_finMixin := FinMixin enum_Z2P.
Canonical Z2_finType := Eval hnf in FinType Z2 Z2_finMixin.
Definition ab_finMixin := FinMixin enum_abP.
Canonical ab_finType := Eval hnf in FinType ab ab_finMixin.

Definition p1_d(x:Z2)(y:ab):Z2 :=
match x,y with
|zero,a ⇒ zero
|zero,b ⇒ one
|one,a ⇒ one
|one,b ⇒ zero
end.
Definition p1 := Automaton Z2_finType ab_finType zero p1_d [set one].
Compute accept p1 [::b;b;b].
```

第2章　Library AutomatonModule

```
From mathcomp Require Import all_ssreflect.

Structure automaton{state symbol:finType}:= Automaton {
  init : state;
  delta : state → symbol → state;
  final : {set state}
}.

Fixpoint dstar{state symbol:finType}(delta:state→symbol→state)
  (q:state)(str:seq symbol):state :=
match str with
|nil ⇒ q
|h::str' ⇒ dstar delta (delta q h) str'
end.

Definition accept{state symbol:finType}(M:@automaton state symbol)
  (str:seq symbol):bool := dstar (delta M) (init M) str“in final M.

Definition accepts{state symbol:finType}(M:@automaton state symbol)
  (l:seq (seq symbol)):seq (seq symbol):=
[seq str←l|accept M str].

Lemma dstarLemma {state symbol : finType}(delta:state→symbol→state)(q:state)
(s t:seq symbol):dstar delta q (s++t) = dstar delta (dstar delta q s) t.
Proof. move:q;by elim:s;[—move⇒a s' H;simpl]. Qed.
```

第3章　Library myLemma

```
From mathcomp Require Import all_ssreflect.

Lemma lesub (m n:nat):m ≤ n ↔ (m - n = 0).
Proof. split;[move/(subnBl_leq 0);by rewrite subn0—].
move:m;elim:n;[move⇒m;rewrite subn0⇒H;by rewrite H—].
move⇒n H;by case;[—move⇒m;rewrite subSS;move/H]. Qed.

Fixpoint filter_option{T:Type}(s:seq (option T)):seq T:=
match s with
|nil ⇒ nil
|Some t::s' ⇒ t::(filter_option s')
|None::s' ⇒ filter_option s'
end.

Lemma bool_eqsplit (a b:bool):(a=b)¡-¿(a↔b).
Proof. split;[by move⇒H;rewrite H—];case;have t:true;[done—];
case:a;case:b;[done| | |done];[move⇒H|move⇒H' H];by move:(H t). Qed.

Lemma map_f' {t1 t2 t3:eqType}(f:t1→t2→t3)(l1:list t1)(l2:list t2)(x1:t1)(x2:t2):
x1 “in l1→x2 “in l2→f x1 x2 “in [seq f x y|x←l1,y←l2].
Proof. move⇒H H1;move:H;elim:l1;[done—];simpl;move⇒a l H2;rewrite
in_cons;
rewrite mem_cat;case H3:(x1==a);[rewrite (eqP H3);move⇒H4{H4};move:H1;
case:(f a x2 “in [seq f x y | x ← l, y ← l2]);
[by rewrite Bool.orb_true_r|rewrite Bool.orb_false_r;apply map_f]—];
case:(f x1 x2 “in [seq f a y | y ← l2]);
[by rewrite Bool.orb_true_l|by rewrite!Bool.orb_false_l]. Qed.

Fixpoint language(n:nat)(symbol:finType):seq (seq symbol):=
match n with
—0 ⇒ [::nil]
|S n' ⇒ [seq s::l|l←language n' symbol,s←enum symbol]
end.

Fixpoint language' (n:nat)(symbol:finType):seq (seq symbol):=

match n with
```

```
—0 ⇒ nil
|S n' ⇒ (language' n' symbol)++(language n symbol)
end.
Lemma language'nil(n:nat)(symbol:finType):
all(fun p⇒p!=nil)(language' n symbol).
Proof.
elim:n.
done.
move⇒n H.
rewrite/=all_cat H/==¿{H}.
elim:(language n symbol).
done.
move⇒a l H{n}.
rewrite/=all_cat H Bool.andb_true_r.
elim:(enum symbol).
done.
move⇒b{}l{}H.
by rewrite/=.
Qed.
Lemma languagelength(n:nat)(symbol:finType):
all(fun p⇒size p==n)(language n symbol).
Proof.
elim:n.
done.
move⇒n.
rewrite/=.
elim:(language n symbol).
done.
move⇒a l H.
simpl.
move/andP.
case.
move⇒H1/H=¿{}H.
rewrite all_cat H Bool.andb_true_r=¿{l H}.
elim:(enum symbol).
done.
move⇒b l H.
by rewrite/=H Bool.andb_true_r eqSS/eqP H1.
Qed.
```

```
Lemma language'length(n:nat)(symbol:finType):
all(fun p=¿0¡size p≤n)(language' n symbol).
Proof.
elim:n.
done.
move⇒n H.
rewrite/=all_cat.
apply/andP.
split.
apply/sub_all/H.
rewrite/subpred⇒x{H}.
case H:(size x≤n).
by rewrite(leqW H).
by case x.
move:(languagelength n.+1 symbol)=¿{}H.
apply/sub_all/H.
rewrite/subpred⇒x/eqP{}H.
by rewrite H leqnn.
Qed.
Lemma languagelemma{V:finType}(s:seq V):s“in (language (size s) V).
Proof. by elim:s;[—move⇒a l H;simpl;apply/map_f';[—apply/mem_enum]].
Qed.
Lemma language'lemma{f:finType}(s:seq f)(n:nat):
s≠nil → size s ≤ n → s “in language' n f.
Proof.
move⇒H/subnKC⇒H1.
rewrite-H1=¿{H1}.
elim:(n - size s).
rewrite addn0.
rewrite/language'.
case_eq(size s).
move:H.
by case s.
move⇒n0 H1.
rewrite mem_cat.
apply/orP.
right.
rewrite-H1.
apply/languagelemma.
```

```
move=¿{H}n H.
rewrite addnS.
simpl.
rewrite mem_cat.
apply/orP.
by left.
Qed.

Lemma fin_index{f:finType}(a:f):index a (enum f) ¡ #—f|.
Proof.
rewrite cardE.
have:a“in (enum f).
apply/mem_enum.
elim:(enum f).
done.
move⇒f0 ef H.
rewrite in_cons.
move/orP.
case.
simpl.
move/eqP⇒af0.
rewrite-af0=¿{f0 af0}.
rewrite (_:a==a=true);[done|by apply/eqP].
move/H=¿{}H.
simpl.
by case:(f0 == a).
Qed.

Lemma fin_zip_neq{f:finType}(x y:seq f):x≠y→size x=size y→
all(fun p⇒p“in zip(enum f)(enum f))(zip x y)=false.
Proof.
move:y.
elim:x.
by case.
move⇒a x H.
case.
done.
move⇒b y H1.
have{}H1:a≠b∨x≠y.
case_eq(a==b)=¿/eqP ab;case_eq(x==y)=¿/eqP xy;subst;by [—right|left|right].
destruct H1.
```

```
suff H2:(a, b) “in zip (enum f) (enum f)=false;[by rewrite/=H2—].
elim:(enum f).
done.
move⇒a0 l H1.
rewrite/=in_cons H1 Bool.orb_false_r.
apply/eqP.
move=¿[H2 H3].
by subst.
move:H0.
move/H=¿{}H.
move=¿[]/H=¿{}H.
by rewrite/=H Bool.andb_false_r.
Qed.

Lemma filter_nil{e:eqType}(l:seq e)(P:e→bool):
[seq a¡-[seq b ← l|P b]—˜˜P a]=nil.
Proof. by elim:l;[—move⇒a l H;simpl;case H1:(P a);[rewrite/=H1—]].
Qed.

Lemma eq_mem_cons{e:eqType}(a:e)(l1 l2:seq e):l1 =i l2 → a::l1 =i a::l2.
Proof.
rewrite/eq_mem⇒H x.
rewrite!in_cons.
apply/orP.
case:(x==a);simpl.
by left.
move:(H x).
case:(x “in l2)=¿{}H;rewrite H.
by right.
by case.
Qed.

Lemma eq_mem_cat{e:eqType}(x y z w:seq e):x=i y→z=i w→x++z=i
y++w.
Proof.
rewrite/eq_mem⇒H H1 x0.
move:(H x0)(H1 x0)=¿{}H{}H1.
rewrite!mem_cat.
by destruct (x0“in z),(x0“in x),(x0“in y).
Qed.

Lemma filter_option_cat{T:Type}(x y:seq (option T)):
```

```
filter_option(x++y)=filter_option x++filter_option y.
Proof.
elim:x.
done.
move⇒a x H.
simpl.
case:a;[—done].
move⇒a.
rewrite cat_cons.
by f_equal.
Qed.

Lemma eq_memS{e:eqType}(x y:seq e):(x =i y)¡-¿(y =i x).
Proof. split;by rewrite/eq_mem. Qed.

Lemma eq_memT{e:eqType}(x y z:seq e):(x =i y)-¿(y =i z)-¿(x=i z).
Proof.
rewrite/eq_mem⇒H H1 x0.
move:(H x0)(H1 x0)=¿{}H.
by rewrite-H.
Qed.

Lemma eq_mem_filter{e:eqType}(x y:seq e)(f:e→bool):(x =i y)-¿
[seq a←x|f a]=i[seq a←y|f a].
Proof.
rewrite/eq_mem⇒H x0.
by rewrite!mem_filter-H.
Qed.

Lemma eq_mem_catC{e:eqType}(x y:seq e):x++y=i y++x.
Proof.
rewrite/eq_mem⇒x0.
rewrite!mem_cat.
by destruct(x0“in x),(x0“in y).
Qed.

Lemma eq_mem_map'{e1 e2 e3:eqType}(x y:seq e1)(z:seq e2)(f:e1→e2→e3):
x=i y-¿[seq f a b|a←x,b←z]=i[seq f a b|a←y,b←z].
Proof.
move⇒H.
elim:z.
have H1:∀(l:seq e1),[seq f a b|a←l,b←nil]=nil;[by elim—].
by rewrite!H1.
```

```
move⇒c z H1.
have H2:∀(l:seq e1),[seq f a b | a ← l, b ← c::z]=i
  [seq f a c|a←l]++[seq f a b|a←l,b←z].
elim.
done.
move⇒a l H2.
remember(c::z)as z'.
rewrite/={1}Heqz'/=.
apply/eq_mem_cons.
rewrite catA.
have H3:([seq f a0 c | a0 ← l] ++ [seq f a b | b ← z]) ++
      [seq f a0 b | a0 ← l, b ← z]=i
       ([seq f a b | b ← z]++[seq f a0 c | a0 ← l]) ++
      [seq f a0 b | a0 ← l, b ← z].
apply/eq_mem_cat;[apply/eq_mem_catC|done].
apply/eq_memT;[—apply/eq_memS/H3].
rewrite-catA.
by apply/eq_mem_cat.
apply/eq_memS.
apply/eq_memT.
apply/H2.
apply/eq_memT;[—apply/eq_memS/H2].
apply/eq_mem_cat.
apply/eq_mem_map/eq_memS/H.
apply/eq_memS/H1.
Qed.

Lemma eq_mem_filter_option{e:eqType}(x y:seq(option e)):
x=i y → filter_option x =i filter_option y.
Proof.
rewrite/eq_mem⇒H x0.
move:(H(Some x0))=¿{H}.
elim:x.
case:y.
done.
move⇒b y.
destruct b.
rewrite/=!in_cons.
case_eq(Some x0==Some s);[done|move/eqP⇒H].
have{}H:x0==s=false.
```

```
apply/eqP.
rewrite/not⇒H1.
by subst.
rewrite H/=!in_nil=¿{}H.
rewrite{}H.
elim:y.
done.
move⇒b y H.
destruct b.
rewrite/=!in_cons.
case_eq(Some x0==Some s0).
move/eqP=¿[H1].
by rewrite H1(_:s0==s0).
move/eqP⇒H1.
have{}H1:x0==s0=false.
apply/eqP.
rewrite/not⇒H2.
by subst.
by rewrite H1.
by rewrite/=in_cons.
rewrite!in_nil⇒H.
rewrite{}H/=in_cons(_:Some x0 == None=false)/=;[—done].
elim:y.
done.
move⇒b y H.
destruct b.
rewrite/=!in_cons.
case_eq(Some x0==Some s).
move/eqP=¿[H1].
by rewrite H1(_:s==s).
move/eqP⇒H1.
have{}H1:x0==s=false.
apply/eqP.
rewrite/not⇒H2.
by subst.
by rewrite H1.
by rewrite/=in_cons(_:Some x0==None=false).
move⇒a x H.
destruct a.
```

```
rewrite/=!in_cons.
case_eq(Some x0 == Some s).
move/eqP=¿[H1].
rewrite{}H1(_:s==s)/=;[—done]=¿H1.
rewrite{H}H1.
elim:y.
done.
move⇒b y H.
destruct b.
rewrite/=!in_cons.
case_eq(Some s==Some s0).
move/eqP=¿[H1].
by rewrite H1(_:s0==s0).
move/eqP⇒H1.
have{}H1:s==s0=false.
apply/eqP.
rewrite/not⇒H2.
by subst.
by rewrite H1.
by rewrite/=in_cons.
move/eqP⇒H1.
have{}H1:x0==s=false.
apply/eqP.
rewrite/not⇒H2.
by subst.
by rewrite H1.
by rewrite/=in_cons(_:Some x0==None=false).
Qed.

Lemma add_subABB(m n:nat):m+n-n=m.
Proof.
elim:n.
by rewrite addn0 subn0.
move⇒n H.
by rewrite addnS subSS.
Qed.
Lemma add_subABA(m n:nat):n+m-n=m.
Proof.
elim:n.
by rewrite add0n subn0.
```

```
move⇒n H.
by rewrite addSn subSS.
Qed.
```

```
Lemma nat_compare(n m:nat):{n¡m}+{n=m}+{n¿m}.
Proof. by move:(Compare_dec.lt_eq_lt_dec n m)=¿
[[/ltP—]—/ltP];[left;left|left;right|right]. Qed.
```

第4章　Library StickerModule

```
From mathcomp Require Import all_ssreflect.
Require Import myLemma ProofIrrelevance.

Definition Rho(symbol:finType):=seq(symbol×symbol).

Structure wk{symbol:finType}{rho:Rho symbol} := Wk{
  str : seq (symbol×symbol);
  nilP : str ≠ nil;
  rhoP : all(fun p⇒p“in rho)str
}.
Structure stickyend{symbol:finType}:= Se{
  is_upper : bool;
  end_str : seq symbol;
  end_nilP : end_str ≠ nil
}.
Inductive domino{symbol:finType}{rho:Rho symbol}:=
|null : domino
|Simplex : @stickyend symbol → domino
|WK : @wk symbol rho → domino
|L : @stickyend symbol → @wk symbol rho → domino
|R : @wk symbol rho → @stickyend symbol → domino
|LR : @stickyend symbol → @wk symbol rho → @stickyend symbol →
domino.

Definition wk_eqb{symbol:finType}{rho:Rho symbol}(x y:@wk symbol rho):bool:=
str x == str y.
Lemma eq_wkP{symbol:finType}{rho:Rho symbol}:
Equality.axiom (@wk_eqb symbol rho).
Proof.
move⇒a b;rewrite/wk_eqb;apply/(iffP idP);[—move⇒ab;by rewrite ab].
move/eqP.
destruct a,b.
simpl⇒H.
subst.
f_equal.
```

```
apply/proof_irrelevance.
apply/eq_irrelevance.
Qed.
Canonical wk_eqMixin{f:finType}{rho:Rho f} := EqMixin (@eq_wkP f
rho).
Canonical wk_eqType{symbol:finType}{rho:Rho symbol} :=
  Eval hnf in EqType _ (@wk_eqMixin symbol rho).
Definition end_eqb{symbol:finType}(x y:@stickyend symbol):bool:=
match x,y with
|Se true s1 _,Se true s2 _ ⇒ s1==s2
|Se false s1 _,Se false s2 _ ⇒ s1==s2
|_,_ ⇒ false
end.
Lemma eq_endP{symbol:finType}:Equality.axiom(@end_eqb symbol).
Proof.
move⇒x y;rewrite/end_eqb;apply/(iffP idP).
destruct x,y.
case:is_upper0;case:is_upper1;[—done|done—];
move/eqP⇒H;subst;f_equal;apply/proof_irrelevance.
move⇒H.
subst.
case:y;case⇒H _;by apply/eqP.
Qed.
Canonical end_eqMixin{symbol:finType} := EqMixin (@eq_endP symbol).
Canonical end_eqType{f:finType}:= Eval hnf in EqType _ (@end_eqMixin
f).
Lemma domino_eq_dec{symbol:finType}{rho:Rho symbol}(x y:@domino sym-
bol rho):
{x=y}+{x≠y}.
Proof.
decide equality.
case_eq(s==s0);move/eqP⇒H;by [left|right].
case_eq(w==w0);move/eqP⇒H;by [left|right].
case_eq(w==w0);move/eqP⇒H;by [left|right].
case_eq(s==s0);move/eqP⇒H;by [left|right].
case_eq(s==s0);move/eqP⇒H;by [left|right].
case_eq(w==w0);move/eqP⇒H;by [left|right].
case_eq(s0==s2);move/eqP⇒H;by [left|right].
case_eq(w==w0);move/eqP⇒H;by [left|right].
case_eq(s==s1);move/eqP⇒H;by [left|right].
Qed.
Definition domino_eqb{symbol:finType}{rho:Rho symbol}(x y:@domino
symbol rho):=
match domino_eq_dec x y with |left _⇒ true|_⇒ false end.
Lemma eq_dominoP{symbol:finType}{rho:Rho symbol}:
Equality.axiom (@domino_eqb symbol rho).
Proof. move⇒a b;rewrite/domino_eqb;apply/(iffP idP);
by case:(domino_eq_dec a b). Qed.
Canonical domino_eqMixin{f:finType}{rho:Rho f} := EqMixin (@eq_dominoP
f rho).
Canonical domino_eqType{symbol:finType}{rho:Rho symbol} :=
  Eval hnf in EqType _ (@domino_eqMixin symbol rho).

Lemma cons_nilP{t:Type}(a:t)(l:seq t):a::l≠nil. Proof. done. Qed.
Definition mu_end{symbol:finType}(rho:Rho symbol)(x y:seq symbol):option
wk:=
match zip x y with
|nil ⇒ None
|a::l⇒ match Bool.bool_dec(all(fun p⇒p“in rho)(a::l)) true with
  |left H ⇒ Some{—str:=a::l;nilP:=cons_nilP a l;rhoP := H—}
  |right _⇒ None
  end
end.
Definition zip'{t:Type}(x y:seq t):seq(t×t):=rev(zip(rev x)(rev y)).
Definition mu_end'{symbol:finType}(rho:Rho symbol)(x y:seq symbol):option
wk:=
match zip' x y with
|nil ⇒ None
|a::l⇒ match Bool.bool_dec(all(fun p⇒p“in rho)(a::l)) true with
  |left H ⇒ Some{—str:=a::l;nilP:=cons_nilP a l;rhoP := H—}
  |right _⇒ None
  end
end.

Lemma cat00{t:Type}(x y:seq t):x++y=nil↔x=nil∧y=nil.
Proof. by split;[case:x;case:y|case⇒x' y';rewrite x' y']. Qed.
Lemma mu_end2_nilP{symbol:finType}(x y:@stickyend symbol):
end_str x++end_str y≠nil.
Proof. rewrite cat00;case⇒x'_;by move:(end_nilP x). Qed.
Definition mu_end2{symbol:finType}(x y:@stickyend symbol):option stick-
```

```
yend:=
match x,y with
|Se true _ _,Se true _ _ ⇒Some
{—is_upper:=true;end_str:=end_str x++end_str y;end_nilP:=mu_end2_nilP
x y—}
|Se false _ _,Se false _ _ ⇒Some
{—is_upper:=false;end_str:=end_str x++end_str y;end_nilP:=mu_end2_nilP
x y—}
|_,_⇒None
end.

Lemma mu_nilP{symbol:finType}{rho:Rho symbol}(x y:@wk symbol rho):
str x++str y≠nil.
Proof. rewrite cat00;case⇒x'_;by move:(nilP x). Qed.
Lemma mu_rhoP{symbol:finType}{rho:Rho symbol}(x y:@wk symbol rho):
all(fun p⇒p“in rho)(str x++str y).
Proof. rewrite all_cat;apply/andP;by move:(rhoP x)(rhoP y). Qed.
Definition mu_wk{symbol:finType}{rho:Rho symbol}(x y:@wk symbol rho):=
{—str := (str x ++ str y);nilP := (mu_nilP x y);rhoP := (mu_rhoP x
y)—}.
Notation ”x # y” := (mu_wk x y)(at level 1,left associativity).

Lemma takenil{t:Type}{x y:seq t}:size x¡size y → take(size y-size x)y≠nil.
Proof.
rewrite/not⇒H H1.
have{H1}:size(take(size y - size x)y)=0.
by rewrite H1.
rewrite size_take ltn_subrL.
case:((0 ¡ size x) && (0 ¡ size y)).
move/lesub.
by rewrite leqNgt H.
by destruct y.
Qed.
Lemma dropnil{t:Type}{x y:seq t}:size x¡size y →drop(size x)y≠nil.
Proof.
rewrite/not⇒H H1.
have{H1}:size(drop(size x)y)=0.
by rewrite H1.
by rewrite size_drop-lesub leqNgt H.
Qed.

Definition mu_endr{symbol:finType}(x y:seq symbol):=
match nat_compare(size x)(size y)with
|inleft(left P) ⇒
  Some{—
    is_upper:=false;
    end_str:=take(size y-size x)y;
    end_nilP:=takenil P
  —}
|inleft(right _)=¿ None
|inright P ⇒
  Some{—
    is_upper:=false;
    end_str:=take(size x-size y)x;
    end_nilP:=takenil P
  —}
end.
Definition mu_endl{symbol:finType}(x y:seq symbol):=
match nat_compare(size x)(size y)with
|inleft(left P) ⇒
  Some{—
    is_upper:=false;
    end_str:=drop(size x)y;
    end_nilP:=dropnil P
  —}
|inleft(right _)=¿ None
|inright P ⇒
  Some{—
    is_upper:=false;
    end_str:=drop(size y)x;
    end_nilP:=dropnil P
  —}
end.
Definition mu{symbol:finType}{rho:Rho symbol}(x y:@domino symbol
rho):=
match x,y with
|null,_ ⇒ Some y
|_,null ⇒ Some x
|Simplex s1,WK w2 ⇒ Some (L s1 w2)
|Simplex (Se true l1 P1),L (Se true l2 P2) w2⇒
  match mu_end2(Se _ true l1 P1)(Se _ true l2 P2) with
  |Some s ⇒ Some(L s w2)
  |None ⇒ None
  end
|Simplex (Se false l1 P1),L (Se false l2 P2) w2⇒
  match mu_end2(Se _ false l1 P1)(Se _ false l2 P2) with
  |Some s ⇒ Some(L s w2)
  |None ⇒ None
  end
|Simplex (Se true l1 _),L (Se false l2 _) w2⇒
  match mu_end' rho l1 l2 with
  |Some w ⇒
    match mu_endr l1 l2 with
    |Some s ⇒ Some(L s w#w2)
    |None ⇒ Some(WK w#w2)
    end
  |None ⇒ None
  end
|Simplex (Se false l1 _),L (Se true l2 _) w2⇒
  match mu_end' rho l2 l1 with
  |Some w ⇒
    match mu_endr l2 l1 with
    |Some s ⇒ Some(L s w#w2)
    |None ⇒ Some(WK w#w2)
    end
  |None ⇒ None
  end
|Simplex s1,R w2 r2 ⇒ Some (LR s1 w2 r2)
|Simplex (Se true l1 P1),LR (Se true l2 P2) w2 r2⇒
  match mu_end2(Se _ true l1 P1)(Se _ true l2 P2) with
  |Some s ⇒ Some(LR s w2 r2)
  |None ⇒ None
  end
|Simplex (Se false l1 P1),LR (Se false l2 P2) w2 r2⇒
  match mu_end2(Se _ false l1 P1)(Se _ false l2 P2) with
  |Some s ⇒ Some(LR s w2 r2)
  |None ⇒ None
  end
|Simplex (Se true l1 _),LR (Se false l2 _) w2 r2⇒
  match mu_end' rho l1 l2 with
  |Some w ⇒
    match mu_endr l1 l2 with
    |Some s ⇒ Some(LR s w#w2 r2)
    |None ⇒ Some(WK w#w2)
    end
  |None ⇒ None
  end
|Simplex (Se false l1 _),LR (Se true l2 _) w2 r2⇒
  match mu_end' rho l2 l1 with
  |Some w ⇒
    match mu_endr l2 l1 with
    |Some s ⇒ Some(LR s w#w2 r2)
    |None ⇒ Some(WK w#w2)
    end
  |None ⇒ None
  end
|WK w1,Simplex s2 ⇒ Some (R w1 s2)
|WK w1,WK w2 ⇒ Some (WK w1#w2)
|WK w1,R w2 r2 ⇒ Some (R w1#w2 r2)
|L l1 w1,Simplex s2 ⇒ Some (LR l1 w1 s2)
|L l1 w1,WK w2 ⇒ Some (L l1 w1#w2)
|L l1 w1,R w2 r2 ⇒ Some (LR l1 w1#w2 r2)
|R w1 (Se true r1 P1),Simplex (Se true l2 P2)=¿
  match mu_end2(Se _ true r1 P1)(Se _ true l2 P2) with
  |Some s ⇒ Some(R w1 s)
  |None ⇒ None
  end
|R w1 (Se false r1 P1),Simplex (Se false l2 P2)=¿
  match mu_end2(Se _ false r1 P1)(Se _ false l2 P2) with
  |Some s ⇒ Some(R w1 s)
  |None ⇒ None
  end
|R w1 (Se true r1 _),Simplex (Se false l2 _)=¿
  match mu_end rho r1 l2 with
  |Some w ⇒
    match mu_endr r1 l2 with
    |Some s ⇒ Some(R w1#w s)
    |None ⇒ Some(WK w1#w)
    end
  |None ⇒ None
  end
|R w1 (Se false r1 _),Simplex (Se true l2 _)=¿
  match mu_end rho l2 r1 with
  |Some w ⇒
    match mu_endr l2 r1 with
    |Some s ⇒ Some(R w1#w s)
    |None ⇒ Some(WK w1#w)
    end
  |None ⇒ None
  end
|R w1 (Se true r1 _),L (Se false l2 _) w2 ⇒
  if size r1 == size l2 then
    match mu_end rho r1 l2 with
    |Some w ⇒ Some (WK w1#w#w2)
    |None ⇒ None
    end
  else
        None
|R w1 (Se false r1 _),L (Se true l2 _) w2 ⇒
  if size r1 == size l2 then
    match mu_end rho l2 r1 with
    |Some w ⇒ Some (WK w1#w#w2)
    |None ⇒ None
    end
  else
        None
|R w1 (Se true r1 _),LR (Se false l2 _) w2 r2 ⇒
  if size r1 == size l2 then
    match mu_end rho r1 l2 with
    |Some w ⇒ Some (R w1#w#w2 r2)
    |None ⇒ None
    end
  else
        None
|R w1 (Se false r1 _),LR (Se true l2 _) w2 r2 ⇒
  if size r1 == size l2 then
    match mu_end rho l2 r1 with
    |Some w ⇒ Some (R w1#w#w2 r2)
    |None ⇒ None
    end
  else
      None
|LR l1 w1 (Se true r1 P1),Simplex (Se true l2 P2)=¿
  match mu_end2(Se _ true r1 P1)(Se _ true l2 P2) with
  |Some s ⇒ Some(LR l1 w1 s)
  |None ⇒ None
  end
|LR l1 w1 (Se false r1 P1),Simplex (Se false l2 P2)=¿
  match mu_end2(Se _ false r1 P1)(Se _ false l2 P2) with
  |Some s ⇒ Some(LR l1 w1 s)
  |None ⇒ None
  end
|LR l1 w1 (Se true r1 _),Simplex (Se false l2 _)=¿
  match mu_end rho r1 l2 with
  |Some w ⇒
    match mu_endr r1 l2 with
    |Some s ⇒ Some(LR l1 w1#w s)
    |None ⇒ Some(L l1 w1#w)
    end
  |None ⇒ None
  end
|LR l1 w1 (Se false r1 _),Simplex (Se true l2 _)=¿
  match mu_end rho l2 r1 with
  |Some w ⇒
    match mu_endr l2 r1 with
    |Some s ⇒ Some(LR l1 w1#w s)
    |None ⇒ Some(L l1 w1#w)
    end
  |None ⇒ None
  end
|LR l1 w1 (Se true r1 _),L (Se false l2 _) w2 ⇒
  if size r1 == size l2 then
    match mu_end rho r1 l2 with
    |Some w ⇒ Some (L l1 w1#w#w2)
    |None ⇒ None
    end
  else
```

```
        None
|LR l1 w1 (Se false r1 _),L (Se true l2 _) w2 ⇒
  if size r1 == size l2 then
    match mu_end rho l2 r1 with
    |Some w ⇒ Some (L l1 w1#w#w2)
    |None ⇒ None
    end
  else
        None
|LR l1 w1 (Se true r1 _),LR (Se false l2 _) w2 r2 ⇒
  if size r1 == size l2 then
    match mu_end rho r1 l2 with
    |Some w ⇒ Some (LR l1 w1#w#w2 r2)
    |None ⇒ None
    end
  else
        None
|LR l1 w1 (Se false r1 _),LR (Se true l2 _) w2 r2 ⇒
  if size r1 == size l2 then
    match mu_end rho l2 r1 with
    |Some w ⇒ Some (LR l1 w1#w#w2 r2)
    |None ⇒ None
    end
  else
        None
|_,_ ⇒ None
end.
```

```
Definition mu'{symbol:finType}{rho:Rho symbol}
(x:@domino symbol rho)(y:@domino symbol rho*@domino symbol rho):=
let (d1,d2) := y in
match mu d1 x with
|Some d ⇒ mu d d2
|None ⇒ None
end.
```

```
Definition st_correct{symbol:finType}{rho:Rho symbol}(x:@domino sym-
bol rho):=
match x with
|WK _ ⇒ true
|L _ _ ⇒ true
```

```
|R _ _ ⇒ true
|LR _ _ _ ⇒ true
|_ ⇒ false
end.
Structure sticker{symbol:finType}{rho:Rho symbol}:= Sticker{
  start : seq (@domino symbol rho);
  extend : seq (@domino symbol rho*@domino symbol rho);
  startP : all st_correct start
}.
Open Scope nat_scope.
Definition is_wk{symbol:finType}{rho:Rho symbol}(x:@domino symbol
rho):bool:=
match x with WK _ ⇒ true|_ ⇒ false end.
Fixpoint ss_generate_prime{symbol:finType}{rho:Rho symbol}
(n:nat)(stk:@sticker symbol rho):seq domino:=
match n with
—0 ⇒ start stk
|S n' ⇒
  let A' := ss_generate_prime n' stk in
  let A_wk := [seq a ← A'|is_wk a] in
  let A_nwk := [seq a ← A'—~~ is_wk a] in
  undup(A_wk++filter_option[seq mu' a d|a←A_nwk,d ← (extend stk)])
end.
Definition decode{symbol:finType}{rho:Rho symbol}(d:@domino symbol
rho):=
match d with|WK (Wk w _ _) ⇒ unzip1 w|_ ⇒ nil end.
Definition ss_language_prime{symbol:finType}{rho:Rho symbol}(n:nat)
(stk:@sticker symbol rho):seq (seq symbol) :=
[seq decode d | d ← ss_generate_prime n stk & is_wk d].
Definition mkend{symbol:finType}(b:bool)(a:symbol)(s:seq symbol):stickyend
:=
{—is_upper:=b;end_str:=a::s;end_nilP:=cons_nilP a s—}.
Lemma zip_rhoP{symbol:finType}(s:seq symbol):
all(fun p⇒p“in(zip(enum symbol)(enum symbol)))(zip s s).
Proof.
elim:s.
done.
move⇒a l H.
```

```
rewrite/=H Bool.andb_true_r=¿{l H}.
have:a“in enum symbol.
apply/mem_enum.
elim:(enum symbol).
done.
move⇒b l H.
rewrite/=!in_cons.
move/orP.
case.
move/eqP⇒H1.
subst.
apply/orP.
left.
by apply/eqP.
move/H=¿{}H.
apply/orP.
by right.
Qed.
Lemma cons_zip_nilP{symbol:finType}(a:symbol)(s:seq symbol):
zip (a::s) (a::s) ≠ nil.
Proof. done. Qed.
Definition mkwkzip{symbol:finType}(a:symbol)(s:seq symbol):wk :=
{—str:=zip(a::s)(a::s);nilP:=cons_zip_nilP a s;rhoP:=zip_rhoP(a::s)—}.
Definition mkWK{symbol:finType}(s:seq symbol):option domino:=
match s with
|nil ⇒ None
|a::s’ ⇒ Some(WK(mkwkzip a s’))
end.
```

第5章　Library REG_RSL

```
From mathcomp Require Import all_ssreflect.
Require Import AutomatonModule StickerModule myLemma.
Definition wkaccept{state symbol:finType}(M:@automaton state symbol)
(s:seq symbol):option domino :=
match s with
|a::s'⇒
  if accept M s then
    Some(WK(mkwkzip a s'))
  else
    None
|_ ⇒ None
end.
Definition startDomino{state symbol:finType}(M:@automaton state sym-
bol)
(s:seq symbol):domino :=
let n := (index(dstar(delta M)(init M)s)(enum state) + 1) in
let w := take(size s - n)s in
let r := drop(size s - n)s in
let rho := zip (enum symbol) (enum symbol) in
match w,r with
|a::w',b::r' ⇒ R(mkwkzip a w')(mkend true b r')
|_,_ ⇒ null
end.
Definition extentionDomino{state symbol:finType}(M:@automaton state
symbol)
(s t:seq symbol):domino×domino:=
let s0 := nth (init M) (enum state) (size t - 1) in
let n := index (dstar (delta M) s0 s) (enum state) + 1 in
let w := take(size s - n)s in
let r := drop(size s - n)s in
match t,w,r with
|a::t',b::w',c::r'=¿(null,LR(mkend false a t')(mkwkzip b w')(mkend true c
r'))
|_,_,_ ⇒ (null:@domino symbol (zip(enum symbol)(enum symbol)),null)
end.

Definition stopDomino{state symbol:finType}(M:@automaton state sym-
bol)
(s t:seq symbol):option(domino×domino):=
let s0 := nth (init M) (enum state) (size s - 1) in
match s,t with
|a::s',b::t'⇒
  if (dstar (delta M) s0 t)“in final M then
    Some(null:@domino symbol (zip(enum symbol)(enum symbol)),
      L(mkend false a s')(mkwkzip b t'))
  else
    None
|_,_⇒None
end.

Lemma st_correctP{state symbol:finType}(M:@automaton state symbol):
all st_correct(filter_option[seq wkaccept M s|s←language'(#—state—.+1)symbol]
  ++[seq startDomino M s|s ← language(#—state—.+1)symbol]).
Proof.
rewrite all_cat.
apply/andP.
split.
move:(language'nil #—state—.+1 symbol).
elim:(language' #—state—.+1 symbol).
done.
move⇒a l H.
simpl.
move/andP.
case⇒H1.
move/H=¿{}H.
rewrite{1}/wkaccept.
move:H1.
case:a.
done.
move⇒a l0 _.
by case:(accept M (a::l0)).

move:(languagelength #—state—.+1 symbol).
elim:(language #—state—.+1 symbol).
```

```
done.
move⇒a l H.
rewrite/=.
move/andP.
case=¿/eqP H1.
move/H=¿{}H.
rewrite H Bool.andb_true_r.
move:H1.
case:a.
done.
simpl.
move⇒a{H}l[H1].
rewrite/startDomino/=.
rewrite H1.
remember(dstar (delta M) (delta M (init M) a) l) as s.
case H:(take(#—state—.+1 - (index s(enum state) + 1))(a :: l)).
have:size(take(#—state—.+1-(index s(enum state) + 1))(a :: l))=0.
by rewrite H.
have H2:(0 ¡ index s (enum state) + 1);[by rewrite addn1—].
have H3:(0 ¡ #—state—.+1);[done—].
rewrite size_take/=H1 ltn_subrL H2 H3/=addn1 subSS =¿{H1 H2 H3
Heqs a l}H.
move:(fin_index s).
by rewrite-subn_gt0 H.
rewrite addn1.
case H2:(drop(#—state—.+1 - (index s(enum state)).+1)(a :: l)).
have{H2}:size(drop(#—state—.+1 - (index s(enum state)).+1)(a :: l))=0.
by rewrite H2.
rewrite size_drop/=H1 subSS subSn.
done.
apply/leq_subr.
done.
Qed.
```

```
Definition Aut_to_Stk{state symbol:finType}(M:@automaton state sym-
bol):=
let A1 := filter_option[seq wkaccept M s|s←language'(#—state—.+1)symbol]
in
let A2 := [seq startDomino M s|s ← language(#—state—.+1)symbol] in
let D1 := [seq extentionDomino M s t|s←language(#—state—.+1)symbol,
                                        t←language'(#—state—)symbol]
in
let D2 := filter_option[seq stopDomino M t s|
  s ← language'(#—state—.+1)symbol,t ← language'(#—state—)symbol]
in
{—start:=(A1++A2);extend:=(D1++D2);startP:=st_correctP M—}.

Lemma lang_gen{state symbol:finType}(M:@automaton state symbol)(a:symbol)
(s:seq symbol)(n:nat):(a::s“in (ss_language_prime n (Aut_to_Stk M)))
 = (WK(mkwkzip a s)“in[seq d←ss_generate_prime n (Aut_to_Stk M)—is_wk
d]).
Proof.
apply/bool_eqsplit.
split.
rewrite/ss_language_prime.
elim(ss_generate_prime n (Aut_to_Stk M)).
done.
move⇒a0 l H {n}.
rewrite/=.
case H1:(is_wk a0);[simpl|by move/H=¿{}H].
rewrite!in_cons.
move/orP=¿[/eqP{}H—].
apply/orP.
left.
rewrite/mkwkzip.
move:H.
rewrite/decode{H1}.
case:a0;(try done).
case⇒st ni rh H.
apply/eqP.
f_equal.
apply/eqP.
rewrite/eq_op/=/wk_eqb/=(_:(a,a)::zip s s=zip(a::s)(a::s));[—done].
have H1:unzip1 st=unzip2 st.
move:rh{ni H}.
elim:st.
done.
move⇒a0 l0 H.
destruct a0.
simpl.
```

```
move/andP=¿[]H1/H{}H.
f_equal;[—apply/H].
move:H1.
elim:(enum symbol).
done.
move=¿{}a{}l{}H.
rewrite/=!in_cons.
move/orP=¿[/eqP[H1 H2]—].
by subst.
done.
by rewrite H{2}H1 zip_unzip.
move/H=¿{}H.
apply/orP.
by right.

rewrite/ss_language_prime/mkwkzip.
move/(map_f decode).
by rewrite/=unzip1_zip.
Qed.

Lemma mu'lemma{state symbol:finType}(M:@automaton state symbol)
(s t u:seq symbol):#—state—.+1¡=size s → size t = #—state—.+1 →
u≠nil→
mu' (startDomino M s)(extentionDomino M t u)=
if drop(size s-((index(dstar(delta M)(init M)s)(enum state)).+1))s ==
u then
  Some (startDomino M (s++t))
else
  None.
Proof.
case_eq s;[done|move⇒a0 l0 s';rewrite-s'].
case_eq t;[done|move⇒a1 l1 t';rewrite-t'].
case_eq u;[done|move⇒a2 l2 u';rewrite-u'].
remember ((index(dstar(delta M)(init M)s)(enum state)).+1) as n.
move⇒lens lent unil.
have lens':n¡=#—state—;[rewrite Heqn;apply/fin_index—].
have{}lens':n¡size s;[by apply/(leq_ltn_trans lens')—].
have lens'':n≤size s;[apply/ltnW/lens'—].

rewrite/extentionDomino u'-u'.
case_eq(take(size t -
        (index (dstar (delta M) (nth (init M) (enum state) (size u - 1))
t)
              (enum state) + 1)) t);[move⇒H|move⇒a3 l3 t1].
have{H}:size(take(size t -
          (index (dstar (delta M) (nth (init M) (enum state) (size u - 1))
t)
              (enum state) + 1)) t)=0;[by rewrite H—].
have H:(0 ¡ index (dstar (delta M) (nth (init M) (enum state) (size u -
1)) t)
             (enum state) + 1);[by rewrite addn1—].
have H1:(0 ¡ size t);[by rewrite t'—].
rewrite size_take ltn_subrL H H1/=lent addn1 subSS=¿{H1}H.
move:(fin_index (dstar (delta M) (nth (init M) (enum state) (size u - 1))
t)).
by rewrite-subn_gt0 H.
case_eq(drop(size t -
          (index (dstar (delta M) (nth (init M) (enum state) (size u - 1))
t)
              (enum state) + 1)) t);[move⇒H|move⇒a4 l4 d2].
have:size(drop(size t -
          (index (dstar (delta M) (nth (init M) (enum state) (size u - 1))
t)
              (enum state) + 1)) t)=0;[by rewrite H—].
have{}H:index(dstar (delta M) (nth (init M) (enum state) (size u - 1))
t)
              (enum state) ¡ size t.
rewrite lent ltnS;apply/ltnW/fin_index.
by rewrite size_drop addn1(subKn H).

rewrite/=/startDomino addn1-Heqn.
case_eq(take (size s - n) s);[move⇒H|move⇒a5 l5 t2].
have:size(take(size s - n)s)=0;[by rewrite H—].
have{}H:(0¡n);[by rewrite Heqn—].
have H1:(0¡size s);[by rewrite s'—].
rewrite size_take ltn_subrL H H1/==¿{H1}H.
move:lens';by rewrite-subn_gt0 H.
case_eq(drop (size s - n) s);[move⇒H|move⇒a6 l6 d1;rewrite-d1].
have:size(drop(size s - n)s)=0;[by rewrite H—].
by rewrite size_drop (subKn(ltnW lens')) Heqn.

have cons_zip:∀(T:Type)(a:T)(l:seq T),zip(a::l)(a::l)=(a,a)::zip l l.
done.
```

```
rewrite/mu/mkend/mu_end.
case_eq(drop (size s - n) s == u)=¿/eqP ueq.
rewrite-u'-d1 ueq(_:size u==size u);[rewrite u' cons_zip|by apply/eqP].
remember(Bool.bool_dec(all(in_mem^~ (mem (zip (enum symbol) (enum
symbol))))
            ((a2, a2) :: zip l2 l2)) true)as B.
rewrite-HeqB{HeqB}.
have{}H:all (in_mem^~ (mem (zip (enum symbol) (enum symbol))))
         ((a2, a2) :: zip l2 l2) = true;[rewrite-cons_zip;apply/zip_rhoP—].
destruct B;[f_equal=¿{H}—contradiction].

case_eq((take(size (s ++ t) -
  (index (dstar (delta M) (init M) (s ++ t)) (enum state) + 1))(s ++
t)))
;[move⇒H|move⇒a7 l7 t3].
have:size(take(size (s ++ t) -
           (index (dstar (delta M) (init M) (s ++ t)) (enum state) + 1))
         (s ++ t))=0;[by rewrite H—].
have{}H:0¡index(dstar(delta M)(init M)(s++t))(enum state)+1;
[by rewrite addn1—].
have H1:0 ¡ size (s ++ t).
by rewrite size_cat s'/=addSn.
rewrite size_take ltn_subrL H H1/=size_cat lent addn1 addnS subSS-
addnBA.
by rewrite s'/=addSn.
apply/ltnW/fin_index.

case_eq(drop(size (s ++ t) -
       (index (dstar (delta M) (init M) (s ++ t)) (enum state) + 1))
     (s ++ t));[move⇒H|move⇒a8 l8 d3].
have:size(drop(size (s ++ t) -
           (index (dstar (delta M) (init M) (s ++ t)) (enum state) + 1))
         (s ++ t))=0;[by rewrite H—].
rewrite size_drop subKn addn1.
done.
rewrite size_cat lent.
apply/ltn_addl.
rewrite ltnS.
apply/ltnW/fin_index.

f_equal.
rewrite/mkwkzip/mu_wk/=.
```

```
apply/eqP.
rewrite/eq_op/=/wk_eqb/=.
apply/eqP.
rewrite-!cons_zip-!zip_cat;[—done|done].
rewrite-cons_zip-!cat_cons-catA{cons_zip}.
suff H:((a5::l5)++(a2::l2)++(a3::l3)=(a7::l7)).
by f_equal.

rewrite-t2-u'-ueq-t1-t3 catA cat_take_drop take_cat-ueq(_:
size(s++t)-(index(dstar(delta M)(init M)(s++t))(enum state)+1)¡size
s=false)=¿
{a2 a3 a4 a5 a6 a7 a8 l2 l3 l4 l5 l6 l7 l8 d1 d2 d3 t1 t2 t3 e u'}.
f_equal.
f_equal.
rewrite size_cat-addnBA.
rewrite add_subABA size_drop dstarLemma.
repeat f_equal.
by rewrite(subKn lens'')subn1 Heqn/=nth_index;[—apply/mem_enum].
rewrite addn1 lent ltnS.
apply/ltnW/fin_index.

rewrite size_cat-addnBA.
rewrite ltnNge.
apply/negbF/leq_addr.
rewrite addn1 lent ltnS.
apply/ltnW/fin_index.

rewrite/mkend.
have:a4::l4=a8::l8.
rewrite-d3-d2 drop_cat size_cat (_:size s+size t -
(index(dstar(delta M)(init M)(s++t))(enum state)+1)¡size s=false).
rewrite-addnBA.
rewrite add_subABA dstarLemma-ueq size_drop.
repeat f_equal.
by rewrite(subKn lens'')subn1 Heqn/=nth_index;[—apply/mem_enum].
rewrite addn1 lent ltnS.
apply/ltnW/fin_index.

rewrite-addnBA.
rewrite ltnNge.
apply/negbF/leq_addr.
rewrite addn1 lent ltnS.
apply/ltnW/fin_index.
```

```
move=¿[H H1].
by subst.
case sizeu:(size (a6 :: l6) == size (a2 :: l2));[—done].
have H:zip (a6 :: l6) (a2 :: l2)=(a6,a2)::zip l6 l2;[done—].
remember(Bool.bool_dec
        (all (in_mem^~ (mem (zip (enum symbol) (enum symbol))))
            ((a6, a2) :: zip l6 l2)) true)as B.
rewrite H-HeqB.
have{}H:(all (in_mem^~ (mem (zip (enum symbol) (enum symbol))))
                ((a6, a2) :: zip l6 l2))=false.
move:sizeu.
rewrite-H-d1-u'=¿/eqP sizeu.
by apply/fin_zip_neq.
destruct B.
move:H.
by rewrite e.
done.
Qed.
Lemma mu'lemma2{state symbol:finType}(M:@automaton state symbol)
(s t u:seq symbol)(d:domino×domino): #—state—¡size s →
Some d = (stopDomino M u t)-¿
mu'(startDomino M s)d =
if (drop (size s-(index(dstar(delta M)(init M)s)(enum state)).+1)s)
  ==u then
  mkWK (s++t)
else
  None.
Proof.
move⇒lens.
rewrite/stopDomino.
case_eq u;[done|move⇒a0 l0 u'].
case_eq t;[done|move⇒a1 l1 t'].
case:(dstar (delta M)(nth (init M)(enum state)(size(a0::l0) - 1)) (a1 ::
l1)
      “in final M);[move=¿[d'];rewrite d'{d'}/=/mu/startDomino|done].
case_eq(take(size s-(index (dstar (delta M) (init M) s) (enum state) +
1)) s)
;[move⇒H|move⇒a2 l2 t1].
have:size(take(size s-(index(dstar(delta M)(init M)s)(enum state) + 1))
s)=0.
by rewrite H.
have{}H:(0 ¡ index (dstar (delta M) (init M)s) (enum state) + 1).
by rewrite addn1.
have H1:(0 ¡ size s);[apply/leq_ltn_trans/lens/leq0n—].
rewrite size_take ltn_subrL H H1/={H H1}.
move/lesub/(leq_trans lens).
rewrite leqNgt.
suff H:((index (dstar (delta M) (init M) s) (enum state) ).+1 ¡ #—state—.+1).
by rewrite addn1 H.
apply/fin_index.
case_eq(drop (size s - (index (dstar (delta M) (init M)s)(enum state)+1))s);
[move⇒H|move⇒a3 l3 d1].
have:size(drop(size s-(index(dstar(delta M)(init M)s)(enum state) + 1))
s)=0;
[by rewrite H—].
rewrite size_drop subKn addn1/=.
done.
apply/leq_ltn_trans/lens/ltnW/fin_index.
case_eq s=¿[—a s']s_.
move:lens.
by rewrite s_.
rewrite/mkWK(_:(a :: s') ++ a1 :: l1=a::(s'++a1::l1));[—done].
rewrite-s_.

case_eq(drop(size s-(index(dstar(delta M)(init M)s)(enum state)).+1)s==a0::l0).
rewrite-addn1 d1.
move/eqP⇒ueq.
rewrite/mkend ueq(_:size(a0::l0)==size(a0::l0));[rewrite/mu_end|by apply/eqP].
have H:zip (a0 :: l0) (a0 :: l0)=(a0,a0)::zip l0 l0;[done—].
rewrite H.
remember(Bool.bool_dec
        (all (in_mem^~ (mem (zip (enum symbol) (enum symbol))))
            ((a0, a0) :: zip l0 l0)) true)as B.
rewrite-HeqB.
have{}H:all (in_mem^~ (mem (zip (enum symbol) (enum symbol))))
                ((a0, a0) :: zip l0 l0);[rewrite-H;apply/zip_rhoP—].
destruct B;[f_equal;f_equal=¿{H}—contradiction].
rewrite(_:a0 :: l0 == a0 :: l0);[—by apply/eqP].
f_equal.
f_equal.
apply/eqP.
rewrite/eq_op/=/wk_eqb/=.
apply/eqP.
have H:∀(a:symbol)(l:seq symbol),
  zip (a :: l) (a :: l)=(a,a)::zip l l;[done—].
rewrite-!H-!zip_cat;[—done|done].
rewrite-!H-!cat_cons{H}.
suff H:((a2 :: l2) ++ a0 :: l0) ++ a1 :: l1=(a :: s') ++ a1 :: l1.
by f_equal.
f_equal.
by rewrite-t1-ueq-d1/=cat_take_drop.

move⇒ueq.
rewrite ueq.
move:ueq.
move/eqP⇒ueq.
rewrite/mkend.
case sizeu:(size (a3 :: l3) == size (a0 :: l0));[—done].
rewrite/mu_end.
have H:∀(T:Type)(a b:T)(x y:seq T),zip(a::x)(b::y)=(a,b)::zip x y;[done—].
rewrite H.
remember(Bool.bool_dec
        (all (in_mem^~ (mem (zip (enum symbol) (enum symbol))))
           ((a3, a0) :: zip l3 l0)) true) as B.
rewrite-HeqB.
have H1:(all (in_mem^~ (mem (zip (enum symbol) (enum symbol))))
               ((a3, a0) :: zip l3 l0))=false.
rewrite-H.
move:ueq.
rewrite-addn1 d1⇒ueq.
have{}sizeu:size (a3::l3)=size (a0::l0).
move:sizeu.
by move/eqP.

by apply/fin_zip_neq.

destruct B;[—done].
move:H1.
by rewrite e.
Qed.

Lemma start_extend{state symbol:finType}(M:@automaton state symbol)
```

```
(n:nat):[seq startDomino M s|s ← language (n.+1*(#—state—.+1)) sym-
bol] =i
  [seq d←ss_generate_prime n (Aut_to_Stk M)—~~is_wk d].
Proof.
elim:n.
rewrite/=plusE addn0 map_cat filter_cat.
have H:[seq d ← filter_option
                    ([seq wkaccept M s | s ← language' #—state| symbol]
++
                     [seq wkaccept M s
                        | s ← [seq s :: l
                                  | l ← language #—state| symbol,
                                    s ← enum symbol]])
          | ~~ is_wk d]=nil.
rewrite-map_cat/wkaccept.
elim:(language' #—state| symbol ++ [seq s :: l
                               | l ← language #—state| symbol, s ← enum sym-
bol]).
done.
simpl.
move⇒a l H.
case:a.
done.
move⇒a l0.
by case:(accept M (a :: l0)).
rewrite H cat0s.
elim:[seq s :: l | l ← language #—state| symbol, s ← enum symbol].
done.
move⇒a l{}H.
rewrite/=(_:~~ is_wk (startDomino M a)).
apply/eq_mem_cons/H.
rewrite/startDomino.
case:(take (size a - (index (dstar (delta M) (init M) a) (enum state) +
1)) a).
done.
move⇒a0 l0.
by case:(drop(size a-(index(dstar (delta M) (init M) a) (enum state) +
1)) a).
move⇒n H.
```

```
remember[seq startDomino M s | s ← language (n.+2 × #—state—.+1)
symbol]as l.
remember(Aut_to_Stk M)as ASM.
rewrite/=filter_undup filter_cat filter_nil cat0s.
apply/eq_memT/eq_memS/mem_undup.
have{}H1:extend ASM = [seq extentionDomino M s t
  | s ← language #—state—.+1 symbol,t ← language' #—state| symbol]
++
      filter_option[seq stopDomino M s t
           | t ← language' #—state—.+1 symbol,s ← language' #—state|
symbol].
by rewrite HeqASM.
move:H.
rewrite{}H1{ASM}HeqASM⇒H.
remember[seq extentionDomino M s t | s ← language #—state—.+1 sym-
bol,
                                      t ← language' #—state| symbol]as A.
remember[seq stopDomino M s t | t ← language' #—state—.+1 symbol,
                                       s ← language' #—state| symbol]as
B.
have{}H1:[seq d ← filter_option[seq mu' a d
                             | a ← [seq a ← ss_generate_prime n (Aut_to_Stk
M)
                            | ~~ is_wk a], d ← A ++ filter_option B]— ~~ is_wk
d] =i
             [seq d ← filter_option([seq mu' a d
                             | a ← [seq a ← ss_generate_prime n (Aut_to_Stk
M)
                                    | ~~ is_wk a], d ← A]++[seq mu' a d
                             | a ← [seq a ← ss_generate_prime n (Aut_to_Stk
M)
                                  | ~~ is_wk a],d ← filter_option B])— ~~
is_wk d].
apply/eq_mem_filter/eq_mem_filter_option/mem_allpairs_catr.
apply/eq_memT/eq_memS/H1.
have{H1}H:[seq d ← filter_option([seq mu' a d
                              | a ← [seq a ← ss_generate_prime n (Aut_to_Stk
M)
                                     | ~~ is_wk a], d ← A] ++
                    [seq mu' a d| a ← [seq a ← ss_generate_prime n (Aut_to_Stk
M)
                                  | ˜˜ is_wk a],d ← filter_option B])— ˜˜ is_wk
d]=i
                [seq d ← filter_option([seq mu' a d
       | a ← [seq startDomino M s | s ← language (n.+1 × #—state—.+1)
symbol],
 d ← A] ++[seq mu' a d| a ← [seq startDomino M s |
        s ← language (n.+1 × #—state—.+1) symbol],
        d ← filter_option B])— ˜˜ is_wk d].
apply/eq_mem_filter/eq_mem_filter_option/eq_mem_cat;
apply/eq_mem_map'/eq_memS/H.
apply/eq_memT/eq_memS/H.
rewrite filter_option_cat filter_cat.
have{}H:[seq d←filter_option[seq mu' a d| a ← [seq startDomino M s
                    | s ← language (n.+1 × #—state—.+1) symbol],
                                d ← filter_option B]—˜˜is_wk d]=nil.
have{H}:all(fun p=¿#—state—.+1¡=size p)(language (n.+1 × #—state—.+1)
symbol).
move:(languagelength(n.+1 × #—state—.+1)symbol).
elim:(language (n.+1 × #—state—.+1) symbol).
done.
move⇒a0 l0{}H.
simpl.
case H1:(size a0 == n.+1 × #—state—.+1);[move/H=¿{}H|done].
rewrite{}H Bool.andb_true_r.
move:H1.
move/eqP.
rewrite mulSn⇒H.
rewrite H addSn.
apply/leq_addr.
elim:(language (n.+1 × #—state—.+1) symbol).
done.
move⇒a0 l0{A HeqA}H.
rewrite/=.
case H1:(#—state| ¡ size a0);[move/H=¿{}H|done].
rewrite filter_option_cat filter_cat{}H cats0{B}HeqB.
elim:(language' #—state—.+1 symbol).
done.
```

```
move⇒a1 l1 H.
rewrite/=filter_option_cat map_cat filter_option_cat filter_cat{}H cats0.
elim:(language' #—state| symbol).
done.
move⇒a2 l2 H.
simpl.
case_eq(stopDomino M a2 a1);[move⇒p H2|done].
rewrite/=.
have{}H2:Some p = stopDomino M a2 a1;[done—].
rewrite (mu'lemma2 _ _ _ _ _ H1 H2)/mkWK.
case:(drop(size a0-(index(dstar(delta M)(init M) a0) (enum state)).+1)
a0 ==
                  a2).
case:(a0 ++ a1)=¿[—a l3];apply/H.
apply/H.
rewrite{B HeqB}H cats0{A}HeqA.
have H:l =i [seq startDomino M(s++t)—s←language(n.+1*#—state—.+1)symbol,
                                                  t←language(#—state—.+1)symbol].
rewrite Heql mulSn addnC.
have:∀ m n:nat,[seq startDomino M s|s←language(m+n)symbol]=i
  [seq startDomino M(s++t)—s←language m symbol,t←language n sym-
bol].
move⇒m n0.
suff H:language(m+n0)symbol=i
[seq s++t|s←language m symbol,t←language n0 symbol].
have{}H:[seq startDomino M s | s ← language (m + n0) symbol]=i[seq
  startDomino M s|s¡-[seq s++t|s←language m symbol,t←language n0
symbol]].
apply/eq_mem_map/H.
apply/eq_memT.
apply/H.
elim:(language m symbol).
done.
move⇒a l0{}H.
rewrite/=map_cat.
apply/eq_mem_cat/H.
elim:(language n0 symbol).
done.
move⇒a0 l1{}H.
```

```
rewrite/=.
apply/eq_mem_cons/H.
rewrite/eq_mem⇒s.
apply/bool_eqsplit.
have languagelength2:∀(n:nat)(s:seq symbol),
size s=n→s“in language n symbol.
move⇒n1 s0 H.
rewrite-{n1}H.
elim:s0.
done.
move⇒a l0 H.
simpl.
apply/map_f'/mem_enum/H.
move:(cat_take_drop m s)=¿H.
split⇒H1.
have{}H1:size s = m+n0.
move:H1(languagelength(m+n0)symbol).
elim:(language(m+n0)symbol).
done.
move⇒a l0 H1.
rewrite/=in_cons.
move/orP=¿[/eqP H2—].
subst.
by move/andP=¿[/eqP].
move/H1=¿{}H1/andP[_].
apply/H1.
rewrite-H.
have H2:size(take m s)=m.
rewrite size_take{}H1.
case:n0.
by rewrite addn0;case(m¡m).
move⇒n0.
by rewrite addnS leq_addr.
have{}H1:size(drop m s)=n0.
move:H1.
rewrite-{1}H size_cat H2.
remember(size(drop m s))as d.
elim m.
done.
```

```
move⇒n1 H1.
rewrite!addSn.
by move=¿[].
apply/map_f'/languagelength2/H1/languagelength2/H2.
apply/languagelength2.
move:H1(languagelength m symbol)(languagelength n0 symbol).
elim:(language m symbol).
done.
move⇒a l0 H1.
rewrite/=mem_cat.
move/orP=¿[H2—/H1{}H1]/andP[]/eqP H3.
subst.
move:H2.
elim:(language n0 symbol).
done.
move⇒a0 l H2.
rewrite/=in_cons.
move/orP=¿[/eqP H3 _ /andP[]/eqP H4 _—/H2{}H2].
by rewrite H3 size_cat-H4.
by move/H2=¿{}H2/andP[]/eqP _/H2{}H2.
done.
apply⇒_.

apply/eq_memT.
apply/H.
suff H1:filter_option
                        [seq mu' a d
                            | a ← [seq startDomino M s
                                          | s ← language (n.+1 × #—state—.+1)
symbol],
                               d ← [seq extentionDomino M s t
                                          | s ← language #—state—.+1 symbol,
                                             t ← language' #—state| symbol]]=i
  [seq startDomino M (s ++ t)
    | s ← language (n.+1 × #—state—.+1) symbol,
       t ← language #—state—.+1 symbol].
have{}H1:[seq d ← filter_option
                        [seq mu' a d
                            | a ← [seq startDomino M s
                                          | s ← language (n.+1 × #—state—.+1)
symbol],
                                    d ← [seq extentionDomino M s t
                                                | s ← language #—state—.+1 symbol,
                                                  t ← language’ #—state| symbol]]
  | ~~ is_wk d]=i[seq d ← [seq startDomino M (s ++ t)
   | s ← language (n.+1 × #—state—.+1) symbol,
     t ← language #—state—.+1 symbol]
        | ~~ is_wk d].
apply/eq_mem_filter/H1.
apply/eq_memT/eq_memS/H1.
move=¿{l Heql H H1}x.
repeat f_equal.
remember(language #—state—.+1 symbol)as l.
elim:(language (n.+1 × #—state—.+1) symbol).
done.
move⇒a0 l0{}H.
rewrite/=filter_cat.
f_equal;[move{Heql H}—done].
elim:l.
done.
move⇒a l H.
rewrite/=(_:~~ is_wk (startDomino M (a0 ++ a)));[by f_equal—]=¿{H
l0 l x n}.
rewrite/startDomino.
by case:(take
          (size (a0 ++ a) -
           (index (dstar (delta M) (init M) (a0 ++ a)) (enum state) + 1))
          (a0 ++ a));case:(drop
          (size (a0 ++ a) -
           (index (dstar (delta M) (init M) (a0 ++ a)) (enum state) + 1))
          (a0 ++ a)).
move:{l Heql H}(languagelength(n.+1 × #—state—.+1)symbol).
elim:(language (n.+1 × #—state—.+1) symbol).
done.
move⇒s l H.
remember(language #—state—.+1 symbol)as t.
rewrite/=filter_option_cat.
move/andP=¿[]/eqP H1/H{}H.
have{}H1:#—state—¡size s;[by rewrite H1 mulSn addSn leq_addr—].
```

```
apply/eq_mem_cat;[—done].
rewrite{H t}Heqt.
move:(languagelength#—state—.+1symbol).
elim:(language#—state—.+1symbol).
done.
move⇒t l0 H.
rewrite/=map_cat filter_option_cat.
move/andP=¿[]/eqP H2/H{}H.
have H3:startDomino M (s ++ t) :: [seq startDomino M (s ++ t0) | t0
← l0]=
[::startDomino M (s ++ t)] ++ [seq startDomino M (s ++ t0) | t0 ← l0].
done.
rewrite{}H3.
apply/eq_mem_cat;[—apply/H].
have:drop(size s-(index(dstar(delta M)(init M)s)(enum state)).+1)s“in
language’ #—state| symbol.
have{}H:0¡size(drop(size s-(index(dstar(delta M)(init M)s)(enum state)).+1)s).
rewrite size_drop subKn.
done.
apply/ltn_trans/H1/fin_index.
have{}H1:size(drop(size s-(index(dstar(delta M)(init M)s)(enum state)).+1)s)
¡=#—state|.
rewrite size_drop subKn;[—apply/ltn_trans/H1];apply/fin_index.
suff H3:drop (size s - (index (dstar (delta M) (init M) s) (enum state)).+1)s
  “in language’
  (size(drop(size s-(index(dstar(delta M)(init M)s)(enum state)).+1)s))symbol.
rewrite-(subnKC H1).
elim:(#—state| -size
       (drop(size s - (index (dstar (delta M) (init M) s) (enum state)).+1)s)).
by rewrite addn0.
move=¿{}n{H1 H2 H3}H.
by rewrite addnS/=mem_cat H.
move:H{H1 H2 n t l l0}.
remember(drop(size s-(index(dstar(delta M)(init M)s)(enum state)).+1)
s)as l.
move{s Heql}.
destruct l.
done.
move⇒_.
```

```
suff:s :: l “in language (size (s :: l)) symbol.
case:(size (s :: l)).
done.
move⇒n.
rewrite/=mem_cat⇒H.
apply/orP.
by right.
elim:(s :: l).
done.
move⇒a{s}l H.
rewrite/=.
apply/map_f'/mem_enum/H.
have:all(fun p⇒p!=nil)(language' #—state| symbol).
elim:#—state|.
done.
move=¿{}n{}H.
rewrite/=all_cat H/=.
elim:(language n symbol).
done.
move⇒a{}l{}H.
rewrite/=all_cat H.
by elim:(enum symbol).
elim:(language' #—state| symbol).
done.
move⇒u{l0 H}l H.
rewrite/=in_cons.
move/andP=¿[]/eqP H3 H4.
rewrite(mu'lemma M _ _ _ H1 H2 H3){H3 n}.
have H3:[:: startDomino M (s ++ t)]=i
(startDomino M (s ++ t))::[:: startDomino M (s ++ t)].
move⇒x.
rewrite!in_cons.
by case:(x == startDomino M (s ++ t)).
move/orP=¿[H5—/H{}H].
rewrite{}H5.
case_eq(drop(size s-(index(dstar(delta M)(init M) s)(enum state)).+1)
s“in l).
move/(H H4)=¿{}H.
apply/eq_memT/eq_memS/H3/eq_mem_cons/H.
```

```
move:{H H3}H4.
elim:l.
done.
move⇒a l H.
rewrite/=in_cons=¿/andP[]/eqP H3/H{}H.
rewrite(mu'lemma M _ _ _ H1 H2 H3).
by case:(drop(size s-(index(dstar(delta M)(init M)s)(enum state)).+1)s
== a).
case:(drop(size s-(index (dstar (delta M) (init M) s) (enum state)).+1)
s ==u).
apply/eq_memT/eq_memS/H3/eq_mem_cons/H/H4.
apply/H/H4.
Qed.

Lemma start_stop{state symbol:finType}(M:@automaton state symbol)(n:nat):
[seq d←ss_generate_prime n.+1 (Aut_to_Stk M)—is_wk d]=i
[seq d←ss_generate_prime n (Aut_to_Stk M)—is_wk d] ++
filter_option[seq wkaccept M s|s←
[seq s++t|s ←language(n.+1*#—state—.+1)symbol,t←language'#—state—.+1symbol]].
Proof.
remember(#—state—.+1)as k.
simpl.
remember[seq extentionDomino M s t
     | s ← language #—state—.+1 symbol,t ← language' #—state| sym-
bol]as A.
rewrite(_:[seq extentionDomino M s t
            | s ← [seq s :: l| l ← language #—state| symbol,s ← enum
symbol],
                                        t ← language' #—state| symbol]=A);[—done].
remember[seq stopDomino M t s | s ← language' #—state—.+1 symbol,
                                          t ← language' #—state| symbol] as B.
rewrite(_:[seq stopDomino M t s | s ← language' #—state| symbol ++
                 [seq s :: l| l ← language #—state| symbol, s ← enum symbol],
                                          t ← language' #—state| symbol]=B);[—done].
apply/eq_memT.
apply/eq_mem_filter/mem_undup.
rewrite filter_cat filter_id.
apply/eq_mem_cat;[done—].
apply/eq_memT.
apply/eq_mem_filter/eq_mem_filter_option/eq_mem_map'/eq_memS/start_extend.
```

```
apply/eq_memT.
apply/eq_mem_filter/eq_mem_filter_option/mem_allpairs_catr.
rewrite filter_option_cat filter_cat(_:[seq a ← filter_option
                [seq mu' x y | x ← [seq startDomino M s
            | s ← language (n.+1 × #—state—.+1) symbol],y ← A]— is_wk
a]=nil).
rewrite cat0s-Heqk.
move:(languagelength(n.+1 × k)symbol).
elim:(language (n.+1 × k) symbol).
done.
move⇒a l H/andP[sizea]/H{}H.
rewrite/=map_cat!filter_option_cat filter_cat.
apply/eq_mem_cat/H.
rewrite{A HeqA H B l}HeqB-Heqk.
case_eq a=¿[H4|s l a_].
move:sizea.
by rewrite H4 Heqk.
rewrite-a_.
move:(language'nil k symbol).
elim:(language' k symbol).
done.
move⇒a0 l0 H/andP[a0nil]/H{}H.
rewrite/=filter_option_cat map_cat filter_option_cat filter_cat.
case_eq(wkaccept M (a ++ a0))=¿[d—]H2.
rewrite-cat1s.
apply/eq_mem_cat/H.
have:drop(size a-(index(dstar(delta M)(init M)a)(enum state)).+1)a“in
language' #—state| symbol.
apply/language'lemma.
rewrite/not=¿{}H.
have:size(drop(size a-(index(dstar(delta M)(init M)a)(enum state)).+1)
a)=0.
by rewrite H.
rewrite size_drop/=subKn.
done.
rewrite(eqP sizea)Heqk mulSn addSn ltnS.
apply/leq_trans/leq_addr/ltnW/fin_index.
rewrite size_drop/=subKn.
apply/fin_index.
```

```
rewrite(eqP sizea)Heqk mulSn addSn ltnS.
apply/leq_trans/leq_addr/ltnW/fin_index.
move:(language'nil#—state|symbol).
elim:(language' #—state| symbol).
done.
move⇒a1 l1{}H/andP[]a1nil/H{}H/orP[H3—/H{}H];rewrite/=.
case_eq(stopDomino M a1 a0).
move⇒p H1.
have{}sizea:#—state—¡size a;[by rewrite(eqP sizea)Heqk leq_addr—].
have{}H1:Some p = stopDomino M a1 a0;[done—].
rewrite/=(mu'lemma2 _ _ _ _ _ sizea H1)H3/mkWK.
have{}H2:WK (mkwkzip s (l ++ a0))=d.
move:H2.
rewrite/wkaccept a_ cat_cons.
by case:(accept M (s :: l ++ a0))=¿[[]—].
rewrite a_ cat_cons/=H2-a_.
move:H.
rewrite(eqP H3).
case H4:(a1 “in l1)=¿H.
apply/eq_memT.
apply/eq_mem_cons.
by apply/H.
move⇒x.
rewrite!in_cons.
by case:(x == d).
apply/eq_mem_cons.
move:H4{H}.
elim:l1.
done.
move⇒a2 l2 H.
rewrite in_cons.
case H4:(a1 == a2);[done|move=¿/H{}H].
rewrite/=.
case_eq(stopDomino M a2 a0)=¿[p0—]H5.
have{}H5:Some p0 = stopDomino M a2 a0;[done—].
by rewrite/=(mu'lemma2 _ _ _ _ _ sizea H5)(eqP H3)H4.
done.
have{}H2:accept M(a++a0).
move:H2.
```

```
rewrite/wkaccept a_ cat_cons.
by case(accept M (s :: l ++ a0)).
rewrite{1}/stopDomino.
have{}H3:size a1 = (index (dstar (delta M) (init M) a) (enum state)).+1.
rewrite-(eqP H3)size_drop subKn.
done.
rewrite(eqP sizea)Heqk mulSn addSn ltnS.
apply/leq_trans/leq_addr/ltnW/fin_index.
rewrite H3 subn1/=nth_index;[—apply/mem_enum].
move:H2.
rewrite/accept dstarLemma⇒H2.
rewrite H2.
by destruct a1,a0.

case_eq(stopDomino M a1 a0)=¿[p—]H1.
have{}sizea:#—state—¡size a;[by rewrite(eqP sizea)Heqk leq_addr—].
have{}H1:Some p = stopDomino M a1 a0;[done—].
rewrite/=(mu'lemma2 _ _ _ _ _ sizea H1).
case H3:(drop(size a -
      (index (dstar (delta M) (init M) a) (enum state)).+1) a ==a1);[—apply/H].
rewrite/mkWK a_ cat_cons/=.
have{}H2:WK (mkwkzip s (l ++ a0))=d.
move:H2.
rewrite/wkaccept a_ cat_cons.
by case:(accept M (s :: l ++ a0))=¿[[]—].
rewrite H2-a_.
apply/eq_memT.
apply/eq_mem_cons/H.
move⇒x.
rewrite!in_cons.
by case(x==d).
apply/H.

rewrite-(cat0s(filter_option
[seq wkaccept M s0 | s0 ← [seq a ++ t | t ← l0]])).
apply/eq_mem_cat/H.
have:drop(size a-(index(dstar(delta M)(init M)a)(enum state)).+1)a“in
language’ #—state| symbol.
apply/language'lemma.
rewrite/not=¿{}H.
have:size(drop(size a-(index(dstar(delta M)(init M)a)(enum state)).+1)
a)=0.
by rewrite H.
rewrite size_drop/=subKn.
done.
rewrite(eqP sizea)Heqk mulSn addSn ltnS.
apply/leq_trans/leq_addr/ltnW/fin_index.
rewrite size_drop/=subKn.
apply/fin_index.
rewrite(eqP sizea)Heqk mulSn addSn ltnS.
apply/leq_trans/leq_addr/ltnW/fin_index.
move:(language'nil#—state|symbol).
elim:(language' #—state| symbol).
done.
move⇒a1 l1{}H/andP[]a1nil/H{}H/orP[H1—/H{}H].
have{}H2:dstar(delta M)(nth(init M)(enum state)(size a1-1))a0“in fi-
nal M=false.
rewrite-(eqP H1)size_drop subKn.
rewrite subn1/=nth_index.
move:H2.
rewrite/wkaccept a_ cat_cons.
case_eq(accept M (s :: l ++ a0));[done—].
by rewrite/accept-cat_cons-a_ dstarLemma.
apply/mem_enum.
rewrite(eqP sizea)Heqk mulSn addSn ltnS.
apply/leq_trans/leq_addr/ltnW/fin_index.
rewrite/={1}/stopDomino H2.
destruct a1 ,a0;[done|done|done—].
move:H.
rewrite (eqP H1).
case H3:(s0::a1 “in l1).
move⇒H.
by apply/H.
move⇒_.
move:H3.
elim:l1.
done.
move⇒a2 l2 H.
rewrite in_cons.
case H3:(s0::a1 == a2);[done—]=¿/H{}H.
```

```
simpl.
case_eq(stopDomino M a2 (s1 :: a0))=¿[p—]H4.
have{}sizea:#—state—¡size a;[by rewrite(eqP sizea)Heqk leq_addr—].
have{}H4:Some p = stopDomino M a2 (s1 :: a0);[done—].
by rewrite/=(mu'lemma2 _ _ _ _ _ sizea H4)(eqP H1)H3.
done.
rewrite/=.

case_eq(stopDomino M a1 a0)=¿[p—]H4.
have{}sizea:#—state—¡size a;[by rewrite(eqP sizea)Heqk leq_addr—].
have{}H4:Some p = stopDomino M a1 a0;[done—].
rewrite/=(mu'lemma2 _ _ _ _ _ sizea H4).
case H1:(drop(size a-(index(dstar(delta M)(init M)a)(enum state)).+1)a==a1).
move:H4.
rewrite/stopDomino-{2}(eqP H1)size_drop subKn.
rewrite subn1/=nth_index.
move:H2.
rewrite/wkaccept a_ cat_cons/accept-cat_cons-a_ dstarLemma.
case:(dstar (delta M) (dstar (delta M) (init M) a) a0 “in final M).
done.
by destruct a1,a0.
apply/mem_enum.
apply/ltn_trans/sizea/fin_index.
apply/H.
apply/H.

rewrite{A B HeqB}HeqA-Heqk.
move:(languagelength(n.+1*k)symbol).
elim:(language (n.+1 × k) symbol).
done.
move⇒a l H/andP[]H1/H{}H.
have{}H1:#—state—¡size a.
by rewrite(eqP H1)Heqk mulSn addSn leq_addr.
rewrite/=filter_option_cat filter_cat{}H cats0.
move:(languagelength k symbol).
elim:(language k symbol).
done.
move⇒a0 l0 H/andP[]H2/H{}H.
have{}H2:size a0 = #—state—.+1.
by rewrite-Heqk-(eqP H2).
rewrite/=map_cat filter_option_cat filter_cat{n k Heqk}H cats0.
```

```
move:(language'nil#—state|symbol).
elim:(language' #—state| symbol).
done.
move⇒a1 l1 H/andP[]H3/H{}H.
move:H3=¿/eqP H3.
rewrite/={H1 H2 H3}(mu'lemma _ _ _ _ H1 H2 H3).
case H1:(drop(size a -
          (index (dstar (delta M) (init M) a) (enum state)).+1) a ==
a1).
rewrite/=/startDomino.
by case:(take
          (size (a ++ a0) -
           (index (dstar (delta M) (init M) (a ++ a0)) (enum state) +
1))
          (a ++ a0));case:(drop
          (size (a ++ a0) -
           (index (dstar (delta M) (init M) (a ++ a0)) (enum state) +
1))
          (a ++ a0)).
apply/H.
Qed.
```

```
Lemma accept_gen{state symbol:finType}(M:@automaton state symbol)(n:nat):
[seq d←ss_generate_prime n (Aut_to_Stk M)—is_wk d] =i
filter_option[seq wkaccept M s|s←language'(n.+1*#—state—.+1)symbol].
Proof.
elim:n=¿[—n H].
rewrite mul1n/=filter_cat.
have H:[seq d ← [seq startDomino M s| s ← [seq s :: l
               | l ← language #—state| symbol,s ← enum symbol]]—is_wk
d]=nil.
elim:[seq s :: l| l ← language #—state| symbol, s ← enum symbol].
done.
move⇒a l H.
rewrite/={1}/startDomino.
by case:(take(size a-(index(dstar (delta M) (init M) a) (enum state) +
1)) a);
case:(drop (size a - (index(dstar (delta M) (init M) a) (enum state) +
1)) a).
rewrite{}H cats0.
```

```
elim:(language' #—state| symbol ++[seq s :: l
                                  | l ← language #—state| symbol,s ← enum symbol]).
done.
move⇒a l H.
rewrite/=.
case_eq(wkaccept M a)=¿[d—];[—done].
rewrite/={1}/wkaccept.
destruct a;[done—].
case:(accept M (s :: a));[—done].
move=¿[H1].
rewrite-H1/=.
apply/eq_mem_cons/H.

apply/eq_memT.
apply/start_stop.
apply/eq_memT.
apply/eq_mem_cat.
apply/H.
done.
remember#—state—.+1as k=¿{Heqk H}.
rewrite-filter_option_cat-map_cat.
apply/eq_mem_filter_option/eq_mem_map.

move⇒x.
rewrite{M}mem_cat.
have H:∀(n:nat)(x:seq symbol),x“in language' n symbol=(0¡size x≤n).
move⇒m x0.
apply/bool_eqsplit.
split.
move:(language'nil m symbol)(language'length m symbol).
elim:(language' m symbol);[done—].
move⇒a l H/andP[]H1/H{}H/andP[]H2/H{}H/orP[/eqP H3—/H{}H];[—done].
by subst.
move=¿/andP[]H H1.
have{}H:x0≠nil;[by destruct x0—].
apply/language'lemma/H1/H.

rewrite!H.
have H1:(x“in [seq s ++ t| s ← language (n.+1 × k) symbol,
                         t ← language' k symbol])=(n.+1*k¡size x≤n.+2*k).
apply/bool_eqsplit.
split.
```

```
move:(languagelength(n.+1*k)symbol).
elim:(language (n.+1 × k) symbol).
done.
move⇒a l{}H1/andP[]/eqP H2/H1{}H1.
rewrite/=mem_cat=¿/orP[—/H1{}H1];[—apply/H1].
move:(language'length k symbol).
elim:(language' k symbol).
done.
move⇒a0 l0{}H1/andP[]H3/H1{}H1/orP[/eqP H4—/H1{}H1];[—apply/H1].
rewrite{}H4 size_cat{}H2.
move:H3=¿/andP[]{}H1 H2.
apply/andP.
split.
by rewrite-{1}(addn0(n.+1*k)) ltn_add2l.
by rewrite(mulSn n.+1)addnC leq_add2r.
move=¿/andP[]H1 H2.
have H3:size(take(n.+1*k)x)=n.+1*k.
by rewrite size_take H1.
have{H1 H2}:0¡size(drop(n.+1*k)x)¡=k.
apply/andP.
split.
move:H1.
rewrite-{1}(cat_take_drop(n.+1*k)x)size_cat.
case:(size (drop(n.+1*k)x)).
by rewrite H3 addn0 ltnn.
done.
move:H2.
by rewrite-{1}(cat_take_drop(n.+1*k)x)size_cat H3(mulSn(n.+1))addnC
leq_add2r.
rewrite-H=¿{}H.
have{}H3:take (n.+1 × k) x“in language (n.+1 × k) symbol.
by rewrite-{2}H3 languagelemma.
by rewrite-{1}(cat_take_drop(n.+1*k)x)map_f'.
rewrite{H}H1(ltnNge(n.+1 × k)).
case H:(0 ¡ size x).
move{H}.
case H:(size x ≤ n.+1 × k).
have{}H:size x ≤ n.+2*k.
rewrite mulSn.
```

```
apply/leq_trans/leq_addl/H.
by rewrite H.
by case:(size x ≤ n.+2 × k).
have{}H:size x = 0;[by destruct x—].
by rewrite H.
Qed.

Theorem REG_RSL{state symbol:finType}(M:@automaton state symbol)(s:seq
symbol)
(m:nat):s ≠ nil → ∃ n:nat , n ≤ m →
accept M s = (s“in(ss_language_prime m (Aut_to_Stk M))).
Proof.
destruct s as[—a s];[done—]=¿_.
apply ex_intro with (size s)=¿H.
rewrite lang_gen accept_gen.
have{H}:a::s“in language' (m.+1 × #—state—.+1) symbol.
apply/language'lemma.
done.
rewrite/=mulnS addSn ltnS.
apply/leq_trans/leq_addr/H.
elim:(language' (m.+1 × #—state—.+1) symbol).
done.
move⇒a0 l0 H.
rewrite in_cons.
case_eq(a :: s == a0)=¿[/eqP H1 _|H1/H{}H].
subst.
move:H.
case H:(a :: s “in l0).
rewrite/=.
case H1:(accept M (a :: s))=¿H2.
by rewrite in_cons(_:(WK (mkwkzip a s) == WK (mkwkzip a s))).
by apply/H2.
move⇒_.
move:H.
simpl.
case H:(accept M (a :: s)).
by rewrite in_cons(_:(WK (mkwkzip a s) == WK (mkwkzip a s))).
elim:l0.
done.
move⇒a0 l0 H1.
```

```
rewrite in_cons.
case H2:(a :: s == a0).
done.
move=¿/H1{}H1.
rewrite/={1}/wkaccept.
destruct a0.
apply/H1.
case H3:(accept M (s0 :: a0));[—apply/H1].
rewrite in_cons-{}H1 Bool.orb_false_r.
case_eq(WK (mkwkzip a s) == WK (mkwkzip s0 a0));[—done].
remember(mkwkzip a s)as A.
remember(mkwkzip s0 a0)as B.
move/eqP=¿[]/eqP.
rewrite/eq_op/=/wk_eqb{H A}HeqA{B}HeqB/mkwkzip/=.
have cons_zip:∀(t:Type)(a:t)(s:seq t),(a,a)::zip s s = zip(a::s)(a::s).
done.
move⇒H.
exfalso.
move:H.
rewrite!cons_zip=¿/eqP H.
have{}H:a::s = s0::a0.
have:unzip1(zip(a::s)(a::s)) = unzip1(zip(s0::a0)(s0::a0)).
by rewrite H.
by rewrite!unzip1_zip.
by move:H2=¿/eqP.
rewrite/={1}/wkaccept.
destruct a0.
done.
case H2:(accept M (s0 :: a0));[—apply/H].
rewrite in_cons-{}H.
case H:(accept M (a :: s)).
by case:(WK (mkwkzip a s) == WK (mkwkzip s0 a0)).
rewrite Bool.orb_false_r.
case_eq(WK (mkwkzip a s) == WK (mkwkzip s0 a0));[—done].
remember(mkwkzip a s)as A.
remember(mkwkzip s0 a0)as B.
move/eqP=¿[]/eqP.
rewrite/eq_op/=/wk_eqb{H A}HeqA{B}HeqB/mkwkzip/=.
have cons_zip:∀(t:Type)(a:t)(s:seq t),(a,a)::zip s s = zip(a::s)(a::s).
```

```
done.
move⇒H.
exfalso.
move:H.
rewrite!cons_zip=¿/eqP H.
have{}H:a::s = s0::a0.
have:unzip1(zip(a::s)(a::s)) = unzip1(zip(s0::a0)(s0::a0)).
by rewrite H.
by rewrite!unzip1_zip.
by move:H1=¿/eqP.
Qed.
```