CoqSticker Module (Ver.0.1)

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 15 日

Email: `hayakawa.ginga.875@s.kyushu-u.ac.jp`

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.
Require Import Ascii String.

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.

Open Scope nat_scope.

Definition eq_string a b :=
  match string_dec a b with left _ ⇒ true | _ ⇒ false end.

Lemma eq_stringP : Equality.axiom eq_string.
Proof. move⇒ ??; apply: (iffP idP); rewrite /eq_string; by case: string_dec.
Qed.

Canonical string_eqMixin := EqMixin eq_stringP.
Canonical string_eqType := Eval hnf in EqType _ string_eqMixin.
```

```
Definition eq_ascii a b :=
  match ascii_dec a b with left _ ⇒ true | _ ⇒ false end.

Lemma eq_asciiP : Equality.axiom eq_ascii.
Proof. move⇒ ??; apply: (iffP idP); rewrite /eq_ascii; by case: ascii_dec.
Qed.

Definition ascii_eqMixin := EqMixin eq_asciiP.
Canonical ascii_eqType := Eval hnf in EqType _ ascii_eqMixin.
Lemma ascii_count_spec :pcancel nat_of_ascii
(fun n:nat⇒Some (ascii_of_nat n)).
Proof. rewrite/pcancel⇒a.
destruct a as [[—][—][—][—][—][—][—][—]]; vm_compute; reflexivity.
Qed.
Definition ascii_countMixin := CountMixin ascii_count_spec.
Definition ascii_choiceMixin := CountChoiceMixin ascii_countMixin.
Canonical ascii_choiceType := Eval hnf in ChoiceType ascii ascii_choiceMixin.
Canonical ascii_countType := Eval hnf in CountType ascii ascii_countMixin.
Definition enum_ascii := [seq ascii_of_nat n|n←List.seq 0 256].
Lemma enum_asciiP :Finite.axiom enum_ascii.
Proof. rewrite/Finite.axiom⇒a.
destruct a as [[—][—][—][—][—][—][—][—]]; vm_compute; reflexivity.
Qed.
Definition ascii_finMixin := FinMixin enum_asciiP.
Canonical ascii_finType := Eval hnf in FinType ascii ascii_finMixin.

Definition p1_d (state:bool_finType)(symbol:ascii_finType) :=
match symbol with
|"a"%char ⇒ state
|"b"%char ⇒ ~~state
|_⇒false
end.

Definition p1 := Automaton bool_finType ascii_finType true p1_d [set
true].
Compute accept p1 [::"a"%char;"b"%char;"a"%char].
```

第3章　Library myLemma

```
From mathcomp Require Import all_ssreflect.
Require Import Bool Ascii String Arith AutomatonModule Recdef.
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.
Lemma filter_cons{T : Type}(p : pred T)(a:T)(s:seq T):
[seq x ← a::s | p x] =
  if p a then a::[seq x←s|p x] else [seq x←s|p x].
Proof. done. Qed.
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 orb_true_r|rewrite orb_false_r;apply map_f]—];
case:(f x1 x2 “in [seq f a y | y ← l2]);
[by rewrite orb_true_l|by rewrite!orb_false_l]. Qed.
Lemma map_f_eq{t1 t2:eqType}(f:t1→t2)(l:list t1)(x:t1):
(∀ x y:t1,f x= f y → x = y) → x“in l = (f x “in [seq f i|i←l]).
Proof.
move⇒H.
elim:l.
done.
move⇒a l H1.
simpl.
rewrite!in_cons-H1.
case:(x “in l);[by rewrite!orb_true_r|rewrite!orb_false_r].
```

```
case_eq(f x==f a).
move/eqP/H=¿{}H.
apply/eqP/H.
case xa:(x==a).
rewrite-(eqP xa).
by move/eqP.
done.
Qed.
Lemma map_f'_eq{t1 t2 t3:eqType}(f:t1→t2→t3)(l1:list t1)(l2:list t2)(x1:t1)
(x2:t2):(∀ (x1 x2:t1)(y1 y2:t2),f x1 y1 = f x2 y2 → x1=x2∧y1=y2)-¿
  (x1 “in l1)&&(x2 “in l2)=(f x1 x2 “in [seq f x y|x←l1,y←l2]).
Proof.
move⇒H.
elim:l1.
done.
move⇒a l H1.
simpl.
rewrite in_cons mem_cat-H1=¿{H1}.
case x1a:(x1==a);simpl.
rewrite-(eqP x1a)-map_f_eq=¿{x1a a}.
by case:(x2 “in l2);case:(x1 “in l).
move⇒x y.
move/H.
by case.
case:((x1 “in l) && (x2 “in l2));[by rewrite orb_true_r|rewrite orb_false_r].
elim:l2.
done.
move⇒b{}l H1.
simpl.
rewrite in_cons-H1 orb_false_r.
case_eq(f x1 x2 == f a b);[move/eqP/H|done].
case.
move:x1a.
by move/eqP.
Qed.

Lemma map_length {t1 t2}(f:t1→t2)(l:list t1):
List.length [seq f x|x←l] = List.length l.
Proof. by elim:l;[—move⇒a l H;simpl;f_equal]. Qed.

Function divide{t:Type}(n:nat)(l:seq t){measure size l}:seq(seq t) :=
```

```
match n,l with
—0,_ ⇒ [::l]
|_,nil ⇒ nil
|_,_=¿(take n l)::(divide n (drop n l))
end.
Proof.
move⇒t n l n0 H t0 l0 H1.
rewrite-{2}(cat_take_drop n0.+1 (t0::l0)).
rewrite size_cat.
simpl.
rewrite addSn.
apply/leP/leq_addl.
Qed.
```

```
Open Scope string_scope.
Definition dividestring (n:nat)(s:string):list string :=
match n,s with
|_,"" ⇒ nil
—0,_ ⇒ [::s]
|_,_ ⇒
let m :=
  if (length s) mod n == 0 then
    (length s)/n
  else
    (length s)/n + 1 in
[seq substring (i×n) n s|i ← List.seq 0 m]
end.
```

```
Lemma appendEmp (s:string):s++""=s.
Proof. by elim:s;[—move⇒a s H;simpl;rewrite H]. Qed.
Lemma substringeq (s:string):substring 0 (length s) s = s.
Proof. by elim:s;[—move⇒a s H;simpl;rewrite H]. Qed.
Lemma substringn0 (n:nat)(s:string):substring n 0 s = "".
Proof. move:n;elim:s;[—move⇒a s H];by case. Qed.
Lemma substringlemma1 (n:nat)(s:string):
substring 0 n s ++ substring n (length s - n) s = s.
Proof. move:s;elim n;
[by move⇒s;rewrite substringn0;rewrite subn0;rewrite substringeq—];
by move⇒n0 H;case;[—simpl;move⇒a s;rewrite subSS;f_equal]. Qed.
Lemma substringlemma2 (m n:nat)(s:string):m+n≤length s →
length (substring m n s) = n.
```

Proof. *have substringlength*:(∀ (*n*:*nat*)(*s*:*string*),
$n \leq$ *length s* → *length* (*substring* 0 *n s*) = *n*);
[elim;[by move⇒*s2*;rewrite *substringn0*—];by move⇒*n0 H3*;case;
[—simpl;move⇒*a s2*;move/*ltnSE*⇒*H4*;move:(*H3 s2 H4*)=¿*H5*;by f_equal]—];
by move:*s*;elim:*m*;[rewrite *add0n*;apply/*substringlength*|move⇒*n0 H*;case;
[—simpl;move⇒*a s*;rewrite *addSn*;move/*ltnSE*;apply *H*]]. Qed.
Lemma *substringlemma3* (*s t*:*string*):*substring* 0 (*length s*) (*s*++*t*) = *s*.
Proof. elim:*s*;[apply/*substringn0*|move⇒*a s H*;simpl;by rewrite *H*]. Qed.
Lemma *substringlemma4* (*m n*:*nat*)(*s t*:*string*):
substring (*length s* + *m*) *n* (*s*++*t*) = *substring m n t*.
Proof. move:*m*;elim:*s*;[simpl;move⇒*m*;by rewrite *add0n*|
move⇒*a s H m*;simpl;apply *H*]. Qed.
Lemma *substringlemma5* (*n*:*nat*)(*s t*:*string*):
substring 0 (*length s* + *n*) (*s*++*t*) = *s*++*substring* 0 *n t*.
Proof. by elim:*s*;[—move⇒*a s H*;simpl;rewrite *H*]. Qed.

第4章 Library StickerModule

```
From mathcomp Require Import all_ssreflect.
Require Import Arith 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.

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_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)—}.

Lemma mu_wkA{symbol:finType}{rho:Rho symbol}:associative (@mu_wk sym-
bol rho).
move⇒x y z;apply/eqP;rewrite/mu_wk/=/eq_op/=/wk_eqb/=catA;by apply/eqP.
Qed.
```

```
Notation "x # y" := (mu_wk x y)(at level 1,left associativity).
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 s1,R w2 r2 => Some (LR s1 w2 r2)
|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 _),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 _),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.
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.
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
  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 $mkwkzip\{symbol{:}finType\}(a{:}symbol)(s{:}seq\ symbol){:}wk :=$
$\{—str{:}{=}zip(a{::}s)(a{::}s);nilP{:}{=}cons\_zip\_nilP\ \ a\ \ s;rhoP{:}{=}zip\_rhoP(a{::}s)—\}.$