Detailed clustering results obtained in the SSReflect library

J. Heras and E. Komendantskaya

In this note, we use ML4PG to analyse the SSReflect library. The results presented in this section are an extension of the results presented in [1, Section 3]. We analyse clusters that are produced in the SSReflect library (v.1.4 for Coq 8.4) using the K-means algorithm and the value 5 as granularity parameter, these options produce the best results in our experiments. ML4PG discovers 280 clusters using those parameters. In 45% of those clusters (126 clusters), all the lemmas belong to the same library. We call a cluster *homogeneous* if it contains lemmas and theorems from one library, and *heterogeneous* if it contain objects from different libraries.

1 Homogeneous clusters

From the 126 clusters, we can distinguish the following classification of clusters.

- 36% of the clusters consist of lemmas about related functions.

Cluster 1

```
Lemma map_take s : map (take n0 s) = take n0 (map s).
Lemma map_drop s : map (drop n0 s) = drop n0 (map s).
```

Cluster 2

```
Lemma minnn : idempotent minn.
Proof. by move=> n; apply/minn_idPl. Qed.
Lemma maxnn : idempotent maxn.
Proof. by move=> n; apply/maxn_idPl. Qed.
```

Cluster 3

```
Lemma ltn_sqr m n : (m ^ 2 < n ^ 2) = (m < n).
Proof. by rewrite ltn_exp2r. Qed.
Lemma leq_sqr m n : (m ^ 2 <= n ^ 2) = (m <= n).
Proof. by rewrite leq_exp2r. Qed.
```

Cluster 4

```
Lemma gtn_max m n1 n2 : (m > maxn n1 n2) = (m > n1) && (m >
    n2).
Proof. by rewrite !ltnNge leq_max negb_or. Qed.
Lemma gtn_min m n1 n2 : (m > minn n1 n2) = (m > n1) || (m >
    n2).
Proof. by rewrite !ltnNge leq_min negb_and. Qed.
```

Cluster 5

```
Lemma addnAC : right_commutative addn.
Proof. by move=> m n p; rewrite -!addnA (addnC n). Qed.
Lemma subnAC : right_commutative subn.
Proof. by move=> m n p; rewrite -!subnDA addnC. Qed.
Lemma mulnAC : right_commutative muln.
Proof. by move=> m n p; rewrite -!mulnA (mulnC n). Qed.
```

Cluster 6

```
Lemma ltn_exp2r m n e : e > 0 -> (m ^ e < n ^ e) = (m < n).
Proof.
move=> e_gt0; apply/idP/idP=> [|ltmn].
  rewrite !ltnNge; apply: contra => lemn.
  by elim: e {e_gt0} => // e IHe; rewrite !expnS leq_mul.
by elim: e e_gt0 => // [[|e] IHe] _; rewrite ?expn1 //
    ltn_mul // IHe.
Qed.

Lemma leq_exp2r m n e : e > 0 -> (m ^ e <= n ^ e) = (m <= n)
    .
Proof. by move=> e_gt0; rewrite leqNgt ltn_exp2r // -leqNgt.
     Qed.
```

Cluster 7

```
Lemma maxnACA : interchange maxn maxn.
Proof. by move=> m n p q; rewrite -!maxnA (maxnCA n). Qed.
Lemma minnACA : interchange minn minn.
Proof. by move=> m n p q; rewrite -!minnA (minnCA n). Qed.
```

Cluster 8

```
Lemma maxnK m n : minn (maxn m n) m = m.
Proof. exact/minn_idPr/leq_maxl. Qed.
Lemma maxKn m n : minn n (maxn m n) = n.
```

```
Proof. exact/minn_idPl/leq_maxr. Qed.
Lemma minnK m n : maxn (minn m n) m = m.
Proof. exact/maxn_idPr/geq_minl. Qed.
Lemma minKn m n : maxn n (minn m n) = n.
Proof. exact/maxn_idPl/geq_minr. Qed.
```

Cluster 9

```
Lemma maxn_minr : right_distributive maxn minn.
Proof. by move=> m n1 n2; rewrite !(maxnC m) maxn_minl. Qed.
Lemma minn_maxr : right_distributive minn maxn.
Proof. by move=> m n1 n2; rewrite !(minnC m) minn_maxl. Qed.
```

Cluster 10

```
Lemma leq0n n : 0 <= n. Proof. by []. Qed.
Lemma ltn0Sn n : 0 < n.+1. Proof. by []. Qed.
```

Cluster 11

```
Lemma maxnSS m n : maxn m.+1 n.+1 = (maxn m n).+1.
Proof. by rewrite !maxnE. Qed.
Lemma minnSS m n : minn m.+1 n.+1 = (minn m n).+1.
Proof. by rewrite -(addn_minr 1). Qed.
```

Cluster 12

```
Lemma has_rcons s x : has (rcons s x) = a x || has s.
Proof. by rewrite -cats1 has_cat has_seq1 orbC. Qed.
Lemma all_rcons s x : all (rcons s x) = a x && all s.
Proof. by rewrite -cats1 all_cat all_seq1 andbC. Qed.
```

Cluster 13

```
Lemma has_predC a s : has (predC a) s = ~~ all a s.
Proof. by elim: s => //= x s ->; case (a x). Qed.

Lemma has_predU a1 a2 s : has (predU a1 a2) s = has a1 s ||
    has a2 s.
Proof. by elim: s => //= x s ->; rewrite -!orbA; do !
    bool_congr. Qed.
```

Cluster 14

```
Lemma has_nil : has [::] = false. Proof. by []. Qed.
Lemma all_nil : all [::] = true. Proof. by []. Qed.
```

Cluster 15

```
Lemma mem_seq2 x y1 y2 : (x \in [:: y1; y2]) = xpred2 y1 y2
    x.
Proof. by rewrite !inE. Qed.

Lemma mem_seq3 x y1 y2 y3 : (x \in [:: y1; y2; y3]) = xpred3
     y1 y2 y3 x.
Proof. by rewrite !inE. Qed.

Lemma mem_seq4 x y1 y2 y3 y4 :
  (x \in [:: y1; y2; y3; y4]) = xpred4 y1 y2 y3 y4 x.
Proof. by rewrite !inE. Qed.
```

Cluster 16

```
Lemma mem_head x s : x \in x :: s.
Proof. exact: predU1l. Qed.
Lemma mem_last x s : last x s \in x :: s.
Proof. by rewrite lastI mem_rcons mem_head. Qed.
Lemma mem_belast s y : {subset belast y s <= y :: s}.
Proof. by move=> x ys'x; rewrite lastI mem_rcons mem_behead.
     Qed.
```

Cluster 17

```
Lemma unzip1_zip s t : size s <= size t -> unzip1 (zip s t)
    = s.
Proof. by elim: s t => [|x s IHs] [|y t] //= le_s_t; rewrite
     IHs. Qed.

Lemma unzip2_zip s t : size t <= size s -> unzip2 (zip s t)
    = t.
Proof. by elim: s t => [|x s IHs] [|y t] //= le_t_s; rewrite
     IHs. Qed.
```

Cluster 18

```
Lemma last_cat x s1 s2 : last x (s1 ++ s2) = last (last x s1
    ) s2.
Proof. by elim: s1 x => [|y s1 IHs] x //=; rewrite IHs. Qed.
Lemma belast_cat x s1 s2 :
  belast x (s1 ++ s2) = belast x s1 ++ belast (last x s1) s2.
Proof. by elim: s1 x => [|y s1 IHs] x //=; rewrite IHs. Qed.
```

Cluster 19

```
Lemma has_seq1 x : has [:: x] = a x.
Proof. exact: orbF. Qed.
Lemma all_seq1 x : all [:: x] = a x.
Proof. exact: andbT. Qed.
```

Cluster 20

```
Lemma drop_size s : drop (size s) s = [::].
Proof. by rewrite drop_oversize // leqnn. Qed.
Lemma take_size s : take (size s) s = s.
Proof. by rewrite take_oversize // leqnn. Qed.
```

Cluster 21

```
Lemma rev_cons x s : rev (x :: s) = rcons (rev s) x.
Proof. by rewrite -cats1 -catrevE. Qed.
Lemma rev_cat s t : rev (s ++ t) = rev t ++ rev s.
Proof. by rewrite -catrev_catr -catrev_catl. Qed.
```

Cluster 22

```
Lemma andbCA : left_commutative andb. Proof. by do 3!case.
    Qed.
Lemma orbCA : left_commutative orb. Proof. by do 3!case. Qed
    .
```

Cluster 23

```
Lemma andb_id2l (a b c : bool) : (a -> b = c) -> a && b = a
    && c.
Proof. by case: a; case: b; case: c => // ->. Qed.
Lemma orb_id2l (a b c : bool) : (~~ a -> b = c) -> a || b =
    a || c.
Proof. by case: a; case: b; case: c => // ->. Qed.
```

Cluster 24

```
Lemma andb_id2r (a b c : bool) : (b -> a = c) -> a && b = c
    && b.
Proof. by case: a; case: b; case: c => // ->. Qed.
Lemma orb_id2r (a b c : bool) : (~~ b -> a = c) -> a || b =
    c || b.
Proof. by case: a; case: b; case: c => // ->. Qed.
```

Cluster 25

```
Lemma andTb : left_id true andb. Proof. by []. Qed.
Lemma orFb : left_id false orb. Proof. by []. Qed.
```

Cluster 26

```
Lemma andbACA : interchange andb andb. Proof. by do 4!case.
    Qed.
Lemma orbACA : interchange orb orb. Proof. by do 4!case. Qed
    .
```

Cluster 27

```
Lemma andbT : right_id true andb. Proof. by case. Qed.
Lemma orbF : right_id false orb. Proof. by case. Qed.
```

Cluster 28

```
Lemma equivPifn : (Q -> P) -> (P -> Q) -> if b then ~ Q else
     Q.
Proof. rewrite -if_neg; exact: equivPif. Qed.

Lemma xorPifn : Q \/ P -> ~ (Q /\ P) -> if b then Q else ~ Q
    .
Proof. rewrite -if_neg; exact: xorPif. Qed.
```

Cluster 29

```
Lemma andb_orl : left_distributive andb orb. Proof. by do 3!
    case. Qed.
Lemma orb_andl : left_distributive orb andb. Proof. by do 3!
    case. Qed.
```

Cluster 30

```
Lemma orb_idl (a b : bool) : (a -> b) -> a || b = b.
Proof. by case: a; case: b => // ->. Qed.
Lemma andb_idl (a b : bool) : (b -> a) -> a && b = b.
Proof. by case: a; case: b => // ->. Qed.
```

Cluster 31

```
Lemma andbA : associative andb. Proof. by do 3!case. Qed.
Lemma orbA : associative orb. Proof. by do 3!case. Qed.
```

Cluster 32

```
Lemma andbC : commutative andb. Proof. by do 2!case. Qed.
Lemma orbC : commutative orb. Proof. by do 2!case. Qed.
```

Cluster 33

```
Lemma in1W : {all1 P1} -> {in D1, {all1 P1}}.
Proof. by move=> ? ?. Qed.
Lemma in2W : {all2 P2} -> {in D1 & D2, {all2 P2}}.
Proof. by move=> ? ?. Qed.
Lemma in3W : {all3 P3} -> {in D1 & D2 & D3, {all3 P3}}.
Proof. by move=> ? ?. Qed.
Lemma on1W : allQ1 f -> {on D2, allQ1 f}. Proof. by move=> ?
    ?. Qed.
Lemma on1lW : allQ1l f h -> {on D2, allQ1l f & h}. Proof. by
    move=> ? ?. Qed.
Lemma on2W : allQ2 f -> {on D2 &, allQ2 f}. Proof. by move=>
    ? ?. Qed.
```

Cluster 34

```
Lemma negb_and (a b : bool) : ~~ (a && b) = ~~ a || ~~ b.
Proof. by case: a; case: b. Qed.

Lemma negb_or (a b : bool) : ~~ (a || b) = ~~ a && ~~ b.
Proof. by case: a; case: b. Qed.
```

Cluster 35

```
Lemma andbK a b : a && b || a = a. Proof. by case: a; case:
   b. Qed.
Lemma andKb a b : a || b && a = a. Proof. by case: a; case:
   b. Qed.
Lemma orbK a b : (a || b) && a = a. Proof. by case: a; case:
    b. Qed.
Lemma orKb a b : a && (b || a) = a. Proof. by case: a; case:
    b. Qed.
```

Cluster 36

```
Lemma inW_bij : bijective f -> {in D1, bijective f}.
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.

Lemma onW_bij : bijective f -> {on D2, bijective f}.
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.
```

```
Lemma inT_bij : {in T1, bijective f} -> bijective f.
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.

Lemma onT_bij : {on T2, bijective f} -> bijective f.
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.
```

Cluster 37

```
Lemma implyb_idl (a b : bool) : (~~ a -> b) -> (a ==> b) = b
    .
Proof. by case: a; case: b => // ->. Qed.
Lemma implyb_idr (a b : bool) : (b -> ~~ a) -> (a ==> b) =
    ~~ a.
Proof. by case: a; case: b => // ->. Qed.
```

Cluster 38

```
Lemma ltn_ord (i : ordinal) : i < n. Proof. exact: valP i.
    Qed.
Lemma leq_ord (i : 'I_n) : i <= n'. Proof. exact: valP i.
    Qed.
```

Cluster 39

```
Lemma eq_card0 A : A =i pred0 -> #|A| = 0.
Proof. exact: eq_card_trans card0. Qed.
Lemma eq_cardT A : A =i predT -> #|A| = size (enum T).
Proof. exact: eq_card_trans cardT. Qed.
Lemma eq_card1 x A : A =i pred1 x -> #|A| = 1.
Proof. exact: eq_card_trans (card1 x). Qed.
```

Cluster 40

```
Lemma cardD1 x A : #|A| = (x \in A) + #|[predD1 A & x]|.
Proof.
case Ax: (x \in A); last first.
  by apply: eq_card => y; rewrite !inE /=; case: eqP => //
      ->.
rewrite /= -(card1 x) -cardUI addnC /=.
rewrite [#|predI _ _|]eq_card0 => [|y]; last by rewrite !inE
    ; case: eqP.
by apply: eq_card => y; rewrite !inE; case: eqP => // ->.
Qed.
Lemma cardU1 x A : #|[predU1 x & A]| = (x \notin A) + #|A|.
Proof.
```

```
case Ax: (x \in A).
  by apply: eq_card => y; rewrite inE /=; case: eqP => // ->.
rewrite /= -(card1 x) -cardUI addnC.
rewrite [#|predI _ _|]eq_card0 => [|y /=]; first exact:
    eq_card.
by rewrite !inE; case: eqP => // ->.
Qed.
```

Cluster 41

```
Lemma cast_ord_inj n1 n2 eq_n : injective (@cast_ord n1 n2
    eq_n).
Proof. exact: can_inj (cast_ordK eq_n). Qed.
Lemma rev_ord_inj {n} : injective (@rev_ord n).
Proof. exact: inv_inj (@rev_ordK n). Qed.
```

Cluster 42

```
Lemma andbN b : b && ~~ b = false. Proof. by case: b. Qed.
Lemma orbN b : b || ~~ b = true. Proof. by case: b. Qed.
```

Cluster 43

```
Lemma andNb b : ~~ b && b = false. Proof. by case: b. Qed.
Lemma orNb b : ~~ b || b = true. Proof. by case: b. Qed.
```

Cluster 44

```
Lemma andb_orr : right_distributive andb orb. Proof. by do
    3!case. Qed.
Lemma orb_andr : right_distributive orb andb. Proof. by do
    3!case. Qed.
```

Cluster 45

```
Lemma andbK a b : a && b || a = a. Proof. by case: a; case:
    b. Qed.
Lemma andKb a b : a || b && a = a. Proof. by case: a; case:
    b. Qed.
Lemma orbK a b : (a || b) && a = a. Proof. by case: a; case:
     b. Qed.
Lemma orKb a b : a && (b || a) = a. Proof. by case: a; case:
     b. Qed.
```

- 20% of clusters contain lemmas that follow the same proof structure and that share some common auxiliary results.

Cluster 46

```
Lemma has_map a s : has a (map s) = has (preim f a) s.
Lemma all_map a s : all a (map s) = all (preim f a) s.
Lemma count_map a s : count a (map s) = count (preim f a) s.
```

Cluster 47

```
Lemma addn1 m : m + 1 = m.+1.
Lemma addn2 m : m + 2 = m.+2.
Lemma addn3 m : m + 3 = m.+3.
Lemma addn4 m : m + 4 = m.+4.
```

Cluster 48

```
Lemma leq_sub2r p m n : m <= n -> m - p <= n - p.
Proof.
by move=> le_mn; rewrite leq_subLR (leq_trans le_mn) // -
    leq_subLR.
Qed.
Lemma leq_exp2r m n e : e > 0 -> (m ^ e <= n ^ e) = (m <= n)
    .
Proof. by move=> e_gt0; rewrite leqNgt ltn_exp2r // -leqNgt.
     Qed.
```

Cluster 49

```
 Lemma ltn_mul2l m n1 n2 : (m * n1 < m * n2) = (0 < m) && (n1
      < n2).
Proof. by rewrite lt0n !ltnNge leq_mul2l negb_or. Qed.

Lemma ltn_mul2r m n1 n2 : (n1 * m < n2 * m) = (0 < m) && (n1
      < n2).
Proof. by rewrite lt0n !ltnNge leq_mul2r negb_or. Qed.
```

Cluster 50

```
Lemma eqn_mul2l m n1 n2 : (m * n1 == m * n2) = (m == 0) || (
    n1 == n2).
Proof. by rewrite eqn_leq !leq_mul2l -orb_andr -eqn_leq. Qed
    .

Lemma eqn_mul2r m n1 n2 : (n1 * m == n2 * m) = (m == 0) || (
    n1 == n2).
Proof. by rewrite eqn_leq !leq_mul2r -orb_andr -eqn_leq. Qed
    .
```

Cluster 51

```
Lemma iterSr n f x : iter n.+1 f x = iter n f (f x).
Proof. by elim: n => //= n <-. Qed.
Lemma iter_add n m f x : iter (n + m) f x = iter n f (iter m
     f x).
Proof. by elim: n => //= n ->. Qed.
```

Cluster 52

```
Lemma leq_pred n : n.-1 <= n. Proof. by case: n => /=. Qed.
Lemma leqSpred n : n <= n.-1.+1. Proof. by case: n => /=.
    Qed.
```

Cluster 53

```
Lemma last_rcons x s z : last x (rcons s z) = z.
Proof. by rewrite -cats1 last_cat. Qed.
Lemma cat_rcons x s1 s2 : rcons s1 x ++ s2 = s1 ++ x :: s2.
Proof. by rewrite -cats1 -catA. Qed.
```

Cluster 54

```
Lemma rot_inj : injective (rot n0). Proof. exact (can_inj
    rotK). Qed.
Lemma rotr_inj : injective (@rotr T n0).
Proof. exact (can_inj rotrK). Qed.
```

Cluster 55

```
Lemma eq_find a1 a2 : a1 =1 a2 -> find a1 =1 find a2.
Proof. by move=> Ea; elim=> //= x s IHs; rewrite Ea IHs. Qed
    .

Lemma eq_filter a1 a2 : a1 =1 a2 -> filter a1 =1 filter a2.
Proof. by move=> Ea; elim=> //= x s IHs; rewrite Ea IHs. Qed
    .
```

Cluster 56

```
Lemma size_map s : size (map s) = size s.
Proof. by elim: s => //= x s ->. Qed.
Lemma find_map a s : find a (map s) = find (preim f a) s.
Proof. by elim: s => //= x s ->. Qed.
```

Cluster 57

```
Lemma prefix_subseq s1 s2 : subseq s1 (s1 ++ s2).
Proof. by rewrite -{1}[s1]cats0 cat_subseq ?sub0seq. Qed.

Lemma suffix_subseq s1 s2 : subseq s2 (s1 ++ s2).
Proof. by rewrite -{1}[s2]cat0s cat_subseq ?sub0seq. Qed.
```

Cluster 58

```
Lemma eq_has a1 a2 : a1 =1 a2 -> has a1 =1 has a2.
Proof. by move=> Ea s; rewrite !has_count (eq_count Ea). Qed
    .

Lemma eq_all a1 a2 : a1 =1 a2 -> all a1 =1 all a2.
Proof. by move=> Ea s; rewrite !all_count (eq_count Ea). Qed
    .
```

Cluster 59

```
Lemma size1_zip s t : size s <= size t -> size (zip s t) =
    size s.
Proof. by elim: s t => [|x s IHs] [|y t] //= Hs; rewrite IHs
    . Qed.

Lemma size2_zip s t : size t <= size s -> size (zip s t) =
    size t.
Proof. by elim: s t => [|x s IHs] [|y t] //= Hs; rewrite IHs
    . Qed.
```

Cluster 60

```
Lemma eq_in_count s : {in s, a1 =1 a2} -> count a1 s = count
     a2 s.
Proof. by move/eq_in_filter=> eq_a12; rewrite !count_filter
    eq_a12. Qed.

Lemma eq_in_has s : {in s, a1 =1 a2} -> has a1 s = has a2 s.
Proof. by move/eq_in_filter=> eq_a12; rewrite !has_filter
    eq_a12. Qed.
```

Cluster 61

```
Lemma perm_catAC s1 s2 s3 : perm_eql ((s1 ++ s2) ++ s3) ((s1
     ++ s3) ++ s2).
```

```
Proof. by apply/perm_eqlP; rewrite -!catA perm_cat2l
    perm_catC. Qed.

Lemma perm_catCA s1 s2 s3 : perm_eql (s1 ++ s2 ++ s3) (s2 ++
     s1 ++ s3).
Proof. by apply/perm_eqlP; rewrite !catA perm_cat2r
    perm_catC. Qed.
```

Cluster 62

```
Lemma contraFT (c b : bool) : (~~ c -> b) -> b = false -> c.
Proof. by move/contraR=> notb_c /negbT. Qed.

Lemma contraFN (c b : bool) : (c -> b) -> b = false -> ~~ c.
Proof. by move/contra=> notb_notc /negbT. Qed.

Lemma contraTF (c b : bool) : (c -> ~~ b) -> b -> c = false.
Proof. by move/contraL=> b_notc /b_notc/negbTE. Qed.

Lemma contraNF (c b : bool) : (c -> b) -> ~~ b -> c = false.
Proof. by move/contra=> notb_notc /notb_notc/negbTE. Qed.
```

Cluster 63

```
Lemma canLR_in x y : {in D1, cancel f g} -> y \in D1 -> x =
    f y -> g x = y.
Proof. by move=> fK D1y ->; rewrite fK. Qed.

Lemma canRL_in x y : {in D1, cancel f g} -> x \in D1 -> f x
    = y -> x = g y.
Proof. by move=> fK D1x <-; rewrite fK. Qed.

Lemma canLR_on x y : {on D2, cancel f & g} -> f y \in D2 ->
    x = f y -> g x = y.
Proof. by move=> fK D2fy ->; rewrite fK. Qed.

Lemma canRL_on x y : {on D2, cancel f & g} -> f x \in D2 ->
    f x = y -> x = g y.
Proof. by move=> fK D2fx <-; rewrite fK. Qed.
```

Cluster 64

```
Lemma inW_bij : bijective f -> {in D1, bijective f}.
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.

Lemma onW_bij : bijective f -> {on D2, bijective f}.
```

```
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.

Lemma inT_bij : {in T1, bijective f} -> bijective f.
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.

Lemma onT_bij : {on T2, bijective f} -> bijective f.
Proof. by case=> g' fK g'K; exists g' => * ? *; auto. Qed.
```

Cluster 65

```
Lemma contraL (c b : bool) : (c -> ~~ b) -> b -> ~~ c.
Proof. by case: b => //; case: c. Qed.
Lemma contraR (c b : bool) : (~~ c -> b) -> ~~ b -> c.
Proof. by case: b => //; case: c. Qed.
Lemma contraLR (c b : bool) : (~~ c -> ~~ b) -> b -> c.
Proof. by case: b => //; case: c. Qed.
```

Cluster 66

```
Lemma all_and2 (hP : forall x, [/\ P1 x & P2 x]) : [/\ a P1
    & a P2].
Proof. by split=> x; case: (hP x). Qed.

Lemma all_and3 (hP : forall x, [/\ P1 x, P2 x & P3 x]) :
  [/\ a P1, a P2 & a P3].
Proof. by split=> x; case: (hP x). Qed.

Lemma all_and4 (hP : forall x, [/\ P1 x, P2 x, P3 x & P4 x])
     :
  [/\ a P1, a P2, a P3 & a P4].
Proof. by split=> x; case: (hP x). Qed.

Lemma all_and5 (hP : forall x, [/\ P1 x, P2 x, P3 x, P4 x &
    P5 x]) :
  [/\ a P1, a P2, a P3, a P4 & a P5].
Proof. by split=> x; case: (hP x). Qed.
```

Cluster 67

```
Lemma if_same : (if b then vT else vT) = vT.
Proof. by case b. Qed.

Lemma if_neg : (if ~~ b then vT else vF) = if b then vF else
     vT.
Proof. by case b. Qed.
```

```
Lemma fun_if : f (if b then vT else vF) = if b then f vT
    else f vF.
Proof. by case b. Qed.
```

Cluster 68

```
Lemma introT : P -> b. Proof. exact: introTF true _. Qed.
Lemma introF : ~ P -> b = false. Proof. exact: introTF false
     _. Qed.
Lemma introN : ~ P -> ~~ b. Proof. exact: introNTF true _.
    Qed.
Lemma introNf : P -> ~~ b = false. Proof. exact: introNTF
    false _. Qed.
Lemma introTn : ~ P -> b'. Proof. exact: introTFn true _.
    Qed.
Lemma introFn : P -> b' = false. Proof. exact: introTFn
    false _. Qed.
Lemma elimT : b -> P. Proof. exact: elimTF true _. Qed.
Lemma elimF : b = false -> ~ P. Proof. exact: elimTF false _
    . Qed.
Lemma elimN : ~~ b -> ~P. Proof. exact: elimNTF true _. Qed.
Lemma elimNf : ~~ b = false -> P. Proof. exact: elimNTF
    false _. Qed.
Lemma elimTn : b' -> ~ P. Proof. exact: elimTFn true _. Qed.
Lemma elimFn : b' = false -> P. Proof. exact: elimTFn false
    _. Qed.
```

Cluster 69

```
Lemma subrelUl r1 r2 : subrel r1 (relU r1 r2).
Proof. by move=> *; apply/orP; left. Qed.

Lemma subrelUr r1 r2 : subrel r2 (relU r1 r2).
Proof. by move=> *; apply/orP; right. Qed.
```

Cluster 70

```
Lemma monoLR_in :
    {in aD &, {mono f : x y / aR x y >-> rR x y}} ->
  {in aD & rD, forall x y, rR (f x) y = aR x (g y)}.
Proof. by move=> mf x y hx hy; rewrite -{1}[y]fgK_on // mf.
    Qed.

Lemma monoRL_in :
    {in aD &, {mono f : x y / aR x y >-> rR x y}} ->
  {in rD & aD, forall x y, rR x (f y) = aR (g x) y}.
```

```
Proof. by move=> mf x y hx hy; rewrite -{1}[x]fgK_on // mf.
    Qed.
```

- 13% of clusters consist of theorems that are used in the proofs of other theorems of the same cluster.

Cluster 71

```
Lemma altP : alt_spec b.
Lemma boolP : alt_spec b1 b1 b1.
```

Cluster 72

```
Lemma leq_addr m n : n <= n + m.
Proof. by rewrite -{1}[n]addn0 leq_add2l. Qed.

Lemma leq_addl m n : n <= m + n.
Proof. by rewrite addnC leq_addr. Qed.
```

Cluster 73

```
Lemma leq_maxl m n : m <= maxn m n. Proof. by rewrite
    leq_max leqnn. Qed.
Lemma leq_maxr m n : n <= maxn m n. Proof. by rewrite maxnC
    leq_maxl. Qed.
```

Cluster 74

```
Lemma geq_minl m n : minn m n <= m. Proof. by rewrite
    geq_min leqnn. Qed.
Lemma geq_minr m n : minn m n <= n. Proof. by rewrite minnC
    geq_minl. Qed.
```

Cluster 75

```
Lemma leq_mul2l m n1 n2 : (m * n1 <= m * n2) = (m == 0) || (
    n1 <= n2).
Proof. by rewrite {1}/leq -mulnBr muln_eq0. Qed.
Lemma leq_pmul2l m n1 n2 : 0 < m -> (m * n1 <= m * n2) = (n1
     <= n2).
Proof. by move/prednK=> <-; rewrite leq_mul2l. Qed.
```

Cluster 76

```
Lemma leq_mul2r m n1 n2 : (n1 * m <= n2 * m) = (m == 0) || (
    n1 <= n2).
Proof. by rewrite -!(mulnC m) leq_mul2l. Qed.
Lemma leq_pmul2r m n1 n2 : 0 < m -> (n1 * m <= n2 * m) = (n1
     <= n2).
Proof. by move/prednK <-; rewrite leq_mul2r. Qed.
```

Cluster 77

```
Lemma maxn_idPl {m n} : reflect (maxn m n = m) (m >= n).
Proof. by rewrite -subn_eq0 -(eqn_add2l m) addn0 -maxnE;
    apply: eqP. Qed.
Lemma maxn_idPr {m n} : reflect (maxn m n = n) (m <= n).
Proof. by rewrite maxnC; apply: maxn_idPl. Qed.
```

Cluster 78

```
Lemma rot_rot m n s : rot m (rot n s) = rot n (rot m s).
Proof.
case: (ltnP (size s) m) => Hm.
  by rewrite !(@rot_oversize T m) ?size_rot 1?ltnW.
case: (ltnP (size s) n) => Hn.
  by rewrite !(@rot_oversize T n) ?size_rot 1?ltnW.
by rewrite !rot_add_mod 1?addnC.
Qed.

Lemma rot_rotr m n s : rot m (rotr n s) = rotr n (rot m s).
Proof. by rewrite {2}/rotr size_rot rot_rot. Qed.
```

Cluster 79

```
Lemma mem_rot s : rot n0 s =i s.
Proof. by move=> x; rewrite -{2}(cat_take_drop n0 s) !
    mem_cat /= orbC. Qed.
Lemma mem_rotr (s : seq T') : rotr n0 s =i s.
Proof. by move=> x; rewrite mem_rot. Qed.
```

Cluster 80

```
Lemma perm_rot n s : perm_eql (rot n s) s.
Proof. by move=> /= s2; rewrite perm_catC cat_take_drop. Qed
    .

Lemma perm_rotr n s : perm_eql (rotr n s) s.
Proof. exact: perm_rot. Qed.
```

Cluster 81

```
Lemma size_enum_ord : size (enum 'I_n) = n.
Proof. by rewrite -(size_map val) val_enum_ord size_iota.
    Qed.
Lemma index_enum_ord (i : 'I_n) : index i (enum 'I_n) = i.
Proof.
by rewrite -{1}(nth_ord_enum i i) index_uniq ?(enum_uniq,
    size_enum_ord).
Qed.
```

Cluster 82

```
Lemma ltn_mul2l m n1 n2 : (m * n1 < m * n2) = (0 < m) && (n1
     < n2).
Proof. by rewrite lt0n !ltnNge leq_mul2l negb_or. Qed.
Lemma ltn_pmul2l m n1 n2 : 0 < m -> (m * n1 < m * n2) = (n1
    < n2).
Proof. by move/prednK <-; rewrite ltn_mul2l. Qed.
```

Cluster 83

```
Lemma ltn_mul2r m n1 n2 : (n1 * m < n2 * m) = (0 < m) && (n1
     < n2).
Proof. by rewrite lt0n !ltnNge leq_mul2r negb_or. Qed.
Lemma ltn_pmul2r m n1 n2 : 0 < m -> (n1 * m < n2 * m) = (n1
    < n2).
Proof. by move/prednK <-; rewrite ltn_mul2r. Qed.
```

Cluster 84

```
Lemma sqrnD m n : (m + n) ^ 2 = m ^ 2 + n ^ 2 + 2 * (m * n).
Proof.
rewrite -!mulnn mul2n mulnDr !mulnDl (mulnC n) -!addnA.
by congr (_ + _); rewrite addnA addnn addnC.
Qed.

Lemma sqrnD_sub m n : n <= m -> (m + n) ^ 2 - 4 * (m * n) =
    (m - n) ^ 2.
Proof.
move=> le_nm; rewrite -[4]/(2 * 2) -mulnA mul2n -addnn
    subnDA.
by rewrite sqrnD addnK sqrn_sub.
Qed.
```

Cluster 85

```
Lemma ltn_Pmull m n : 1 < n -> 0 < m -> m < n * m.
Proof. by move=> lt1n m_gt0; rewrite -{1}[m]mul1n ltn_pmul2r
    . Qed.

Lemma ltn_Pmulr m n : 1 < n -> 0 < m -> m < m * n.
Proof. by move=> lt1n m_gt0; rewrite mulnC ltn_Pmull. Qed.
```

Cluster 86

```
Lemma addn_min_max m n : minn m n + maxn m n = m + n.
Proof. by rewrite /minn /maxn; case: ltngtP => // [_|->] //;
     exact: addnC. Qed.

Lemma minnE m n : minn m n = m - (m - n).
Proof. by rewrite -(subnDl n) -maxnE -addn_min_max addnK
    minnC. Qed.
```

- 11% of clusters are formed by "view" lemmas, an important kind of lemmas that are used in SSReflect to apply boolean reflection.

Cluster 87

```
Lemma unit_enumP : Finite.axiom [::tt]. by case. Qed.
Lemma bool_enumP : Finite.axiom [:: true; false]. by case.
    Qed.
```

Cluster 88

```
Lemma maxn_idPr {m n} : reflect (maxn m n = n) (m <= n).
Proof. by rewrite maxnC; apply: maxn_idPl. Qed.
Lemma minn_idPr {m n} : reflect (minn m n = n) (m >= n).
Proof. by rewrite minnC; apply: minn_idPl. Qed.
```

Cluster 89

```
Lemma leqifP m n C : reflect (m <= n ?= iff C) (if C then m
    == n else m < n).
Proof.
rewrite ltn_neqAle; apply: (iffP idP) => [/lte]; last by
    rewrite !lte; case C.
by case C => [/eqP-> | /andP[/negPf]]; split=> //; exact:
    eqxx.
Qed.
Lemma ltngtP m n : compare_nat m n (m < n) (n < m) (m == n).
Proof.
```

```
rewrite ltn_neqAle eqn_leq; case: ltnP; first by constructor
    .
by rewrite leq_eqVlt orbC; case: leqP; constructor; first
    exact/eqnP.
Qed.
```

Cluster 90

```
Lemma ltnP m n : ltn_xor_geq m n (n <= m) (m < n).
Proof. by rewrite -(ltnS n); case: leqP; constructor. Qed.
Lemma posnP n : eqn0_xor_gt0 n (n == 0) (0 < n).
Proof. by case: n; constructor. Qed.
```

Cluster 91

```
Lemma addE : add =2 addn.
Proof. by elim=> //= n IHn m; rewrite IHn addSnnS. Qed.
Lemma expE : exp =2 expn.
Proof. by move=> m [|n] //=; rewrite mul_expE expnS mulnC.
    Qed.
```

Cluster 92

```
Lemma allP s : reflect (forall x, x \in s -> a x) (all a s).
Proof.
elim: s => [|x s IHs]; first by left.
rewrite /= andbC; case: IHs => IHs /=.
  apply: (iffP idP) => [Hx y|]; last by apply; exact:
      mem_head.
  by case/predU1P=> [->|Hy]; auto.
by right=> H; case IHs => y Hy; apply H; exact: mem_behead.
Qed.

Lemma allPn s : reflect (exists2 x, x \in s & ~~ a x) (~~
    all a s).
Proof.
elim: s => [|x s IHs]; first by right=> [[x Hx _]].
rewrite /= andbC negb_and; case: IHs => IHs /=.
  by left; case: IHs => y Hy Hay; exists y; first exact:
      mem_behead.
apply: (iffP idP) => [|[y]]; first by exists x; rewrite ?
    mem_head.
by case/predU1P=> [-> // | s_y not_a_y]; case: IHs; exists y
    .
Qed.
```

Cluster 93

```
Lemma eqseqP : Equality.axiom eqseq.
Proof.
move; elim=> [|x1 s1 IHs] [|x2 s2]; do [by constructor |
    simpl].
case: (x1 =P x2) => [<-|neqx]; last by right; case.
by apply: (iffP (IHs s2)) => [<-|[]].
Qed.
Lemma nthP (T : eqType) (s : seq T) x x0 :
  reflect (exists2 i, i < size s & nth x0 s i = x) (x \in s).
Proof.
apply: (iffP idP) => [|[n Hn <-]]; last by apply mem_nth.
by exists (index x s); [rewrite index_mem | apply nth_index
    ].
Qed.
Lemma perm_eqrP s1 s2 : reflect (perm_eqr s1 s2) (perm_eq s1
     s2).
Proof.
apply: (iffP idP) => [/perm_eqlP eq12 s3| <- //].
by rewrite !(perm_eq_sym s3) eq12.
Qed.
```

Cluster 94

```
Lemma idP : reflect b1 b1.
Proof. by case b1; constructor. Qed.
Lemma idPn : reflect (~~ b1) (~~ b1).
Proof. by case b1; constructor. Qed.
Lemma negP : reflect (~ b1) (~~ b1).
Proof. by case b1; constructor; auto. Qed.
```

Cluster 95

```
Lemma introP : (b -> Q) -> (~~ b -> ~ Q) -> reflect Q b.
Proof. by case b; constructor; auto. Qed.

Lemma iffP : (P -> Q) -> (Q -> P) -> reflect Q b.
Proof. by case: Pb; constructor; auto. Qed.
```

Cluster 96

```
Lemma existsP P : reflect (exists x, P x) [exists x, P x].
Proof. by apply: (iffP pred0Pn) => [] [x]; exists x. Qed.
Lemma exists_eqP f1 f2 :
  reflect (exists x, f1 x = f2 x) [exists x, f1 x == f2 x].
```

```
Proof. by apply: (iffP (existsP _)) => [] [x /eqP]; exists x
    . Qed.
Lemma codomP y : reflect (exists x, y = f x) (y \in codom f)
    .
Proof. by apply: (iffP (imageP _ y)) => [][x]; exists x. Qed
    .
```

Cluster 97

```
Lemma eqfunP f1 f2 : reflect (forall x, f1 x = f2 x) [forall
     x, f1 x == f2 x].
Proof. by apply: (iffP (forallP _)) => eq_f12 x; apply/eqP/
    eq_f12. Qed.
Lemma injectiveP : reflect (injective f) injectiveb.
Proof. apply: (iffP (dinjectiveP _)) => injf x y => [|_ _];
    exact: injf. Qed.
```

Cluster 98

```
Lemma pickP : pick_spec (pick P).
Proof.
rewrite /pick; case: (enum _) (mem_enum P) => [|x s] Pxs /=.
  by right; exact: fsym.
by left; rewrite -[P _]Pxs mem_head.
Qed.
Lemma subsetPn A B :
  reflect (exists2 x, x \in A & x \notin B) (~~ (A \subset B)
      ).
Proof.
rewrite unlock; apply: (iffP (pred0Pn _)) => [[x] | [x Ax
    nBx]].
  by case/andP; exists x.
by exists x; rewrite /= nBx.
Qed.
```

Cluster 99

```
Lemma eqfun_inP D f1 f2 :
  reflect {in D, forall x, f1 x = f2 x} [forall (x | x \in D)
      , f1 x == f2 x].
Proof. by apply: (iffP (forall_inP _ _)) => eq_f12 x Dx;
    apply/eqP/eq_f12. Qed.
Lemma option_enumP : Finite.axiom option_enum.
Proof. by case=> [x|]; rewrite /= count_map (count_pred0,
    enumP). Qed.
```

Cluster 100

```
Lemma properE A B : A \proper B = (A \subset B) && ~~(B \
    subset A).
Proof. by []. Qed.
Lemma codomE : codom f = map f (enum T).
Proof. by []. Qed.
```

- 5% of the clusters contain equivalence lemmas that are proven just by simplification.

Cluster 101

```
Lemma multE : mult = muln. Proof. by []. Qed.
Lemma mulnE : muln = muln_rec. Proof. by []. Qed.
Lemma addnE : addn = addn_rec. Proof. by []. Qed.
Lemma plusE : plus = addn. Proof. by []. Qed.
```

Cluster 102

```
Lemma add1n m : 1 + m = m.+1. Proof. by []. Qed.
Lemma add2n m : 2 + m = m.+2. Proof. by []. Qed.
Lemma add3n m : 3 + m = m.+3. Proof. by []. Qed.
Lemma add4n m : 4 + m = m.+4. Proof. by []. Qed.
```

Cluster 103

```
Lemma minn0 : right_zero 0 minn. Proof. by []. Qed.
Lemma maxn0 : right_zero 0 maxn. Proof. by []. Qed.
```

Cluster 104

```
Lemma factE : factorial = fact_rec. Proof. by []. Qed.
Lemma doubleE : double = double_rec. Proof. by []. Qed.
```

Cluster 105

```
Lemma fact0 : 0`! = 1. Proof. by []. Qed.
Lemma double0 : 0.*2 = 0. Proof. by []. Qed.
```

Cluster 106

```
Lemma expn0 m : m ^ 0 = 1. Proof. by []. Qed.
Lemma expn1 m : m ^ 1 = m. Proof. by []. Qed.
```

- 4% of the clusters consist of lemmas that are solved using analogous lemmas.

Cluster 107

```
Lemma addnAC : right_commutative addn.
Proof. by move=> m n p; rewrite -!addnA (addnC n). Qed.
Lemma subnAC : right_commutative subn.
Proof. by move=> m n p; rewrite -!subnDA addnC. Qed.
Lemma mulnAC : right_commutative muln.
Proof. by move=> m n p; rewrite -!mulnA (mulnC n). Qed.
```

Cluster 108

```
Lemma maxnACA : interchange maxn maxn.
Proof. by move=> m n p q; rewrite -!maxnA (maxnCA n). Qed.
Lemma minnACA : interchange minn minn.
Proof. by move=> m n p q; rewrite -!minnA (minnCA n). Qed.
```

Cluster 109

```
Lemma maxn_minr : right_distributive maxn minn.
Proof. by move=> m n1 n2; rewrite !(maxnC m) maxn_minl. Qed.
Lemma minn_maxr : right_distributive minn maxn.
Proof. by move=> m n1 n2; rewrite !(minnC m) minn_maxl. Qed.
```

Cluster 110

```
Lemma expIn e : e > 0 -> injective (expn^~ e).
Proof. by move=> e_gt1 m n /eqP; rewrite eqn_exp2r // => /
    eqP. Qed.
Lemma expnI m : 1 < m -> injective (expn m).
Proof. by move=> m_gt1 e1 e2 /eqP; rewrite eqn_exp2l // => /
    eqP. Qed.
```

Cluster 111

```
Lemma eqn_mul2l m n1 n2 : (m * n1 == m * n2) = (m == 0) || (
    n1 == n2).
Proof. by rewrite eqn_leq !leq_mul2l -orb_andr -eqn_leq. Qed
    .

Lemma eqn_mul2r m n1 n2 : (n1 * m == n2 * m) = (m == 0) || (
    n1 == n2).
Proof. by rewrite eqn_leq !leq_mul2r -orb_andr -eqn_leq. Qed
    .
```

- Unclear pattern.

Cluster 112

```
Lemma ex_maxn_subproof : exists i, P (m - i).
Proof. by case: exP => i Pi; exists (m - i); rewrite subKn ?
    ubP. Qed.
Lemma mulnA : associative muln.
Proof. by move=> m n p; elim: m => //= m; rewrite mulSn
    mulnDl => ->. Qed.
```

Cluster 113

```
Lemma addnBA m n p : p <= n -> m + (n - p) = m + n - p.
Proof. by move=> le_pn; rewrite -{2}(subnK le_pn) addnA
    addnK. Qed.
Lemma odd_sub m n : n <= m -> odd (m - n) = odd m (+) odd n.
Proof.
by move=> le_nm; apply: (@canRL bool) (addbK _) _; rewrite -
    odd_add subnK.
Qed.
```

Cluster 114

```
 Lemma size_zip s t : size (zip s t) = minn (size s) (size t)
     .
Proof.
by elim: s t => [|x s IHs] [|t2 t] //=; rewrite IHs -add1n
    addn_minr.
Qed.
Lemma reshapeKr sh s : size s <= sumn sh -> flatten (reshape
     sh s) = s.
Proof.
elim: sh s => [[]|n sh IHsh] //= s sz_s; rewrite IHsh ?
    cat_take_drop //.
by rewrite size_drop leq_subLR.
Qed.
```

Cluster 115

```
Lemma count_filter
Lemma has_count
Lemma filter_cat
Lemma has_predC
Lemma has_predU
Lemma all_predC
```

```
Lemma filter_mask
Lemma sumn_cat
Lemma size_flatten
Lemma flatten_cat
Lemma allpairs_cat
```

Cluster 116

```
Lemma size_behead
Lemma nth_nil
Lemma set_nth_nil
Lemma drop0
Lemma take0
Lemma headI
Lemma mask0
Lemma mask1
Lemma mask_cons
Lemma has_mask_cons
Lemma mem_mask_cons
Lemma behead_map
```

Cluster 117

```
Lemma index_map s x : index (f x) (map f s) = index x s.
Proof. by rewrite /index; elim: s => //= y s IHs; rewrite (
    inj_eq Hf) IHs. Qed.
 Lemma nth_iota m n i : i < n -> nth 0 (iota m n) i = m + i.
Proof.
by move/subnKC <-; rewrite addSnnS iota_add nth_cat
    size_iota ltnn subnn.
Qed.
```

Cluster 118

```
Lemma all_pred1_nseq
Lemma catCA_perm_subst
Lemma map_uniq
```

Cluster 119

```
Lemma eq_from_nth
Lemma uniq_size_uniq
Lemma rot_add_mod
```

Cluster 120

Lemma *nth_default*
Lemma *drop_oversize*
Lemma *all_pred1_constant*
Lemma *set_nth_default*
Lemma *nth_map*

Cluster 121

Lemma *sym_right_transitive*
Lemma *subon2*

Cluster 122

Lemma *negb_and*
Lemma *negb_or*
Lemma *andbK*
Lemma *andKb*
Lemma *orbK*
Lemma *orKb*
Lemma *negb_imply*
Lemma *implybE*
Lemma *implyNb*
Lemma *implybN*
Lemma *implybNN*
Lemma *addbN*
Lemma *addNb*
Lemma *app_predE*
Lemma *in_collective*
Lemma *in_simpl*

Cluster 123

Lemma *ifE*
Lemma *implyTb*
Lemma *addTb*
Lemma *unfold_in*
Lemma *mem_simpl*
Lemma *qualifE*
Lemma *forE*

Cluster 124

Lemma *nat_irrelevance*
Lemma *subSnn*
Lemma *lt_irrelevance*

Lemma *ltn_add2r*
Lemma *maxnK*
Lemma *maxKn*
Lemma *minnK*
Lemma *minKn*
Lemma *mulSnr*
Lemma *sqrn_gt0*

Cluster 125

Lemma *eqn_add2r*
Lemma *subnDr*
Lemma *ltnn*
Lemma *leq_subLR*
Lemma *ltn_subRL*
Lemma *geq_max*
Lemma *geq_min*
Lemma *mulnSr*
Lemma *leq_mul2r*
Lemma *expnSr*
Lemma *expnAC*
Lemma *muln2*
Lemma *ltn_double*
Lemma *ltn_Sdouble*
Lemma *doubleMl*
Lemma *doubleMr*
Lemma *mulnn*

Cluster 126

Lemma *cardUI*
Lemma *cardC*
Lemma *card2*
Lemma *disjoint_cat*
Lemma *preim_iinv*
Lemma *leq_image_card*
Lemma *card_unit*
Lemma *card_bool*
Lemma *mem_sub_enum*
Lemma *val_enum_ord*
Lemma *size_enum_ord*
Lemma *index_enum_ord*
Lemma *enum_valP*
Lemma *bump_addl*
Lemma *inord_val*

2 Heterogeneous clusters

In the case of heterogeneous clusters (clusters that include lemmas from different libraries), ML4PG discovers 154 clusters. In this case, the size of the clusters is bigger than in the case of homogeneous clusters; namely, the mean size is 8 lemmas per cluster. The different clusters can be classified as follows.

- 31% of the clusters contain lemmas that state properties applicable to several operators from different libraries. In particular, the operations are: associative, commutative, left_id, right_id, left_zero, right_zero, left_distributive, right_distributive, interchange, idempotent, left_commutative, right_commutative, left_injective, right_injective, injective, cancel, pcancel, ocancel, bijective, self_inverse.

  . Some examples.

Cluster 127

```
Lemma catA s1 s2 s3 : s1 ++ s2 ++ s3 = (s1 ++ s2) ++ s3.
Lemma addnA : associative addn.
```

Cluster 128

```
Lemma add0n : left_id 0 addn. Proof. by []. Qed.
Lemma andTb : left_id true andb. Proof. by []. Qed.
Lemma cat0s s : [::] ++ s = s. Proof. by []. Qed.
Lemma orFb : left_id false orb. Proof. by []. Qed.
```

Cluster 129

```
Lemma andFb : left_zero false andb. Proof. by []. Qed.
Lemma mul0n : left_zero 0 muln. Proof. by []. Qed.
```

Cluster 130

```
Lemma subn0 : right_id 0 addn. Proof. by case. Qed.
Lemma andbT : right_id true andb. Proof. by case. Qed.
emma orbF : right_id false orb. Proof. by case. Qed.
Lemma cats0 s : s ++ [::] = s.
Proof. by elim: s => //= x s ->. Qed.
```

Cluster 131

```
Lemma andbF : right_zero false andb. Proof. by case. Qed.
Lemma muln0 : right_zero 0 muln. Proof. by elim. Qed.
```

Cluster 132

```
Lemma mulnDl : left_distributive muln addn.
Proof. by move=> m1 m2 n; elim: m1 => //= m1 IHm; rewrite -
    addnA -IHm. Qed.
Lemma andb_orl : left_distributive andb orb. Proof. by do 3!
    case. Qed.
```

Cluster 133

```
Lemma mulnDr : right_distributive muln addn.
Proof. by move=> m n1 n2; rewrite !(mulnC m) mulnDl. Qed.
Lemma andb_orr : right_distributive andb orb. Proof. by do
    3!case. Qed.
```

Cluster 134

```
Lemma addnACA : interchange addn addn.
Proof. by move=> m n p q; rewrite -!addnA (addnCA n). Qed.
Lemma mulnACA : interchange muln muln.
Proof. by move=> m n p q; rewrite -!mulnA (mulnCA n). Qed.
Lemma andbACA : interchange andb andb. Proof. by do 4!case.
    Qed.
Lemma orbACA : interchange orb orb. Proof. by do 4!case. Qed
    .
```

Cluster 135

```
Lemma mulnC : commutative muln.
Proof.
by move=> m n; elim: m => [|m]; rewrite (muln0, mulnS) //
    mulSn => ->.
Qed.
Lemma addnC : commutative addn.
Proof. by move=> m n; rewrite -{1}[n]addn0 addnCA addn0. Qed
    .
Lemma orbC : commutative orb. Proof. by do 2!case. Qed.
Lemma andbC : commutative andb. Proof. by do 2!case. Qed.
```

Cluster 136

```
Lemma eq_leq
Lemma perm_eqlE
```

- 27% of the clusters consist of lemmas related to operations over the base case of types. Some examples:

Cluster 137

```
Lemma andTb : left_id true andb.
Lemma orFb : left_id false orb.
Lemma mul0n : left_zero 0 muln.
Lemma sub0n : left_zero 0 subn.
```

Cluster 138

```
Lemma andFb : left_zero false andb. Proof. by []. Qed.
Lemma mul0n : left_zero 0 muln. Proof. by []. Qed.
```

Cluster 139

```
Lemma subn0 : right_id 0 addn. Proof. by case. Qed.
Lemma andbT : right_id true andb. Proof. by case. Qed.
emma orbF : right_id false orb. Proof. by case. Qed.
Lemma cats0 s : s ++ [::] = s.
Proof. by elim: s => //= x s ->. Qed.
```

Cluster 140

```
Lemma andbF : right_zero false andb. Proof. by case. Qed.
Lemma muln0 : right_zero 0 muln. Proof. by elim. Qed.
```

Cluster 141

```
Lemma andb_orl
Lemma orb_andl
Lemma andb_addl
Lemma minn0
```

Cluster 142

```
Lemma is_true_true
Lemma implyFb
Lemma leq0n
Lemma ltn0Sn
Lemma ltnSn
```

Cluster 143

```
Lemma negbT
Lemma negbTE
Lemma negbF
Lemma negbFE
```

```
Lemma negbNE
Lemma if_arg
Lemma ltn_predK
Lemma lt0n_neq0
Lemma neq0_lt0n
```

Cluster 144

```
Lemma implybT
Lemma implyFb
Lemma implybb
Lemma leq0n
Lemma ltn0Sn
```

Cluster 145

```
Lemma has_nil
Lemma all_nil
Lemma expn0
Lemma fact0
Lemma double0
```

Cluster 146

```
Lemma nth_nil
Lemma set_nth_nil
Lemma double_gt0
Lemma double_eq0
```

- 12% of the clusters come from lemmas whose proof rely on the fundamental lemmas.

Cluster 147

```
Lemma rot0 s : rot 0 s = s.
Lemma expn_eq0 m e : (m ^ e == 0) = (m == 0) && (e > 0).
```

Cluster 148

```
Lemma andbb
Lemma orbb
Lemma ltnP
```

Cluster 149

Lemma *ltn_sub2r*
Lemma *ltn_sub2l*
Lemma *leq_pmull*
Lemma *map_id_in*
Lemma *pmap_uniq*

Cluster 150

Lemma *subn_sqr*
Lemma *uniq_catCA*

Cluster 151

Lemma *mem_mem*
Lemma *keyed_predE*

Cluster 152

Lemma *leq_exp2l*
Lemma *allpairs_uniq*

Cluster 153

Lemma *or4P*
Lemma *maxn_minl*
Lemma *allP*
Lemma *allPn*
Lemma *leq_size_perm*
Lemma *rot_rot*
Lemma *subseq_uniqP*

Cluster 154

Lemma *negb_inj*
Lemma *perm_rotr*

Cluster 155

Lemma *implybT*
Lemma *implybb*
Lemma *leqnn*
Lemma *leqnSn*
Lemma *leq_b1*
Lemma *sub0seq*

Cluster 156

Lemma *can_mono*
Lemma *mem_take*
Lemma *mem_mask_rot*

Cluster 157

Lemma *eq_leq*
Lemma *perm_eqlE*

Cluster 158

Lemma *addKn*
Lemma *lastP*

Cluster 159

Lemma *leqif_trans*
Lemma *perm_to_subseq*

Cluster 160

Lemma *sqrn_sub*
Lemma *eq_from_nth*
Lemma *nth_take*
Lemma *uniq_size_uniq*
Lemma *rot_add_mod*

Cluster 161

Lemma *leP*
Lemma *ltn_mul*
Lemma *rotrK*
Lemma *rev_rotr*
Lemma *rot_addn*
Lemma *mem_iota*

Cluster 162

Lemma *ltngtP*
Lemma *all_count*

Cluster 163

```
Lemma prop_congr
Lemma eq_mkseq
```

Cluster 164

```
Lemma leq_mul
Lemma sqrnD
Lemma sqrnD_sub
Lemma take_cat
Lemma filter_pred1_uniq
Lemma nth_uniq
Lemma rev_rot
```

- 9% of the clusters combine lemmas from the libraries about lists and natural numbers – note that the definition of lists and natural numbers is quite similar, both have one base case and a recursive one, so several lemmas are solved applying induction and using the inductive hypothesis.

Cluster 165

```
Lemma catrev_catr s t u : catrev s (t ++ u) = catrev s t ++
    u.
Lemma mulnDl : left_distributive muln addn.
Lemma mem_cat x s1 s2: (x \in s1 ++ s2) = (x \in s1) || (x \
    in s2).
```

Cluster 166

```
Lemma size0nil s : size s = 0 -> s = [::]. Proof. by case: s
    . Qed.
Lemma lt0n_neq0 n : 0 < n -> n != 0. Proof. by case: n. Qed.
Lemma neq0_lt0n n : (n == 0) = false -> 0 < n. Proof. by
    case: n. Qed.
```

Cluster 167

```
Lemma rev_zip s1 s2 :
  size s1 = size s2 -> rev (zip s1 s2) = zip (rev s1) (rev s2
    ).
Proof.
elim: s1 s2 => [|x s1 IHs] [|y s2] //= [eq_sz].
by rewrite !rev_cons zip_rcons ?IHs ?size_rev.
Qed.
Lemma expnM m n1 n2 : m ^ (n1 * n2) = (m ^ n1) ^ n2.
Proof.
elim: n1 => [|n1 IHn]; first by rewrite exp1n.
by rewrite expnD expnS expnMn IHn.
Qed.
```

Cluster 168

```
Lemma nth_nseq m x n : nth (nseq m x) n = (if n < m then x
    else x0).
Proof. by elim: m n => [|m IHm] [|n] //=; exact: IHm. Qed.
Lemma subn_gt0 m n : (0 < n - m) = (m < n).
Proof. by elim: m n => [|m IHm] [|n] //; exact: IHm n. Qed.
```

Cluster 169

```
Lemma uniq_catCA s1 s2 s3 : uniq (s1 ++ s2 ++ s3) = uniq (s2
     ++ s1 ++ s3).
Proof.
by rewrite !catA -!(uniq_catC s3) !(cat_uniq s3) uniq_catC !
    has_cat orbC.
Qed.
Lemma subn_sqr m n : m ^ 2 - n ^ 2 = (m - n) * (m + n).
Proof. by rewrite mulnBl !mulnDr addnC (mulnC m) subnDl !
    mulnn. Qed.
```

Cluster 170

```
Lemma rotS n s : n < size s -> rot n.+1 s = rot 1 (rot n s).
Proof. exact: (@rot_addn 1). Qed.
Lemma ltnW m n : m < n -> m <= n.
Proof. exact: leq_trans. Qed.
```

Cluster 171

```
Lemma leq_double m n : (m.*2 <= n.*2) = (m <= n).
Proof. by rewrite /leq -doubleB; case (m - n). Qed.
Lemma has_filter a s : has a s = (filter a s != [::]).
Proof. by rewrite has_count count_filter; case (filter a s).
     Qed.
```

Cluster 172

```
Lemma uniq_catCA s1 s2 s3 : uniq (s1 ++ s2 ++ s3) = uniq (s2
     ++ s1 ++ s3).
Proof.
by rewrite !catA -!(uniq_catC s3) !(cat_uniq s3) uniq_catC !
    has_cat orbC.
Qed.
Lemma minnE m n : minn m n = m - (m - n).
Proof. by rewrite -(subnDl n) -maxnE -addn_min_max addnK
    minnC. Qed.
```

Cluster 173

```
Lemma muln_eq1 m n : (m * n == 1) = (m == 1) && (n == 1).
Proof. by case: m n => [|[|m]] [|[|n]] //; rewrite muln0.
    Qed.
Lemma nth_last s : nth s (size s).-1 = last x0 s.
Proof. by case: s => //= x s; rewrite last_nth. Qed.
```

Cluster 174

```
Lemma nilP s : reflect (s = [::]) (nilp s).
Proof. by case: s => [|x s]; constructor. Qed.
Lemma maxn_idPl {m n} : reflect (maxn m n = m) (m >= n).
Proof. by rewrite -subn_eq0 -(eqn_add2l m) addn0 -maxnE;
    apply: eqP. Qed.
```

Cluster 175

```
Lemma rot_oversize n s : size s <= n -> rot n s = s.
Proof. by move=> le_s_n; rewrite /rot take_oversize ?
    drop_oversize. Qed.
Lemma odd_sub m n : n <= m -> odd (m - n) = odd m (+) odd n.
Proof.
by move=> le_nm; apply: (@canRL bool) (addbK _) _; rewrite -
    odd_add subnK.
Qed.
```

Cluster 176

```
Lemma nth_behead s n : nth (behead s) n = nth s n.+1.
Proof. by case: s n => [|x s] [|n]. Qed.
Lemma subnDA m n p : n - (m + p) = (n - m) - p.
Proof. by elim: m n => [|m IHm] [|n]; try exact (IHm n). Qed
    .
```

Cluster 177

```
Lemma natnseq0P s : reflect (s = nseq (size s) 0) (sumn s ==
     0).
Proof.
apply: (iffP idP) => [|->]; last by rewrite sumn_nseq.
by elim: s => //= x s IHs; rewrite addn_eq0 => /andP[/eqP->
    /IHs <-].
Qed.
Lemma mulnBl : left_distributive muln subn.
Proof.
```

```
move=> m n [|p]; first by rewrite !muln0.
by elim: m n => // [m IHm] [|n] //; rewrite mulSn subnDl -
    IHm.
Qed.
```

- Unclear pattern.

Cluster 178

```
Lemma leq_sub2r
Lemma leq_exp2r
Lemma drop_size_cat
Lemma zip_rcons
```

Cluster 179

```
Lemma addSnnS
Lemma addn1
Lemma addn2
Lemma addn3
Lemma addn4
Lemma subSKn
Lemma ltnNge
Lemma maxnSS
Lemma minnSS
Lemma ltn_sqr
Lemma leq_sqr
Lemma eqn_sqr
Lemma catrevE
Lemma mem_seq2
Lemma mem_seq3
Lemma mem_seq4
Lemma has_pred1
Lemma size_rotr
Lemma has_rotr
Lemma rotr_uniq
Lemma foldl_cat
Lemma cardE
Lemma cardT
Lemma cardC1
Lemma subsetE
Lemma subset_all
Lemma topredE
```

Cluster 180

Lemma *ltn_predK*
Lemma *lt0n_neq0*
Lemma *neq0_lt0n*
Lemma *size0nil*
Lemma *negbT*
Lemma *negbTE*
Lemma *negbF*
Lemma *negbFE*
Lemma *negbNE*

Cluster 181

Lemma *eqSS*
Lemma *addSn*
Lemma *add1n*
Lemma *add2n*
Lemma *add3n*
Lemma *add4n*
Lemma *subSS*
Lemma *ltnS*
Lemma *ltn0*
Lemma *subn_eq0*
Lemma *iterS*
Lemma *iteriS*
Lemma *mulSn*
Lemma *expn0*
Lemma *expn1*
Lemma *fact0*
Lemma *factS*
Lemma *double0*
Lemma *doubleS*
Lemma *cat0s*
Lemma *cat1s*
Lemma *cat_cons*
Lemma *rcons_cons*
Lemma *last_cons*
Lemma *nth0*
Lemma *has_nil*
Lemma *all_nil*
Lemma *drop_cons*
Lemma *take_cons*
Lemma *eqseq_cons*
Lemma *in_cons*
Lemma *in_nil*
Lemma *cons_uniq*
Lemma *subseq0*

Lemma *map_cons*
Lemma *properE*
Lemma *negb_forall*
Lemma *codomE*
Lemma *lift0*
Lemma *ifE*
Lemma *implyTb*
Lemma *addTb*
Lemma *unfold_in*
Lemma *mem_simpl*
Lemma *qualifE*
Lemma *forE*

Cluster 182

Lemma *prednK*
Lemma *ltnW*
Lemma *rotS*
Lemma *pmap_sub_uniq*
Lemma *eq_card0*
Lemma *eq_cardT*
Lemma *eq_card1*
Lemma *canF_LR*
Lemma *canF_RL*
Lemma *eq_codom*
Lemma *eq_card_sub*
Lemma *widen_ord_proof*
Lemma *eq_card_prod*
Lemma *negbLR*
Lemma *negbRL*
Lemma *introT*
Lemma *introF*
Lemma *introN*
Lemma *introNf*
Lemma *introTn*
Lemma *introFn*
Lemma *elimT*
Lemma *elimF*
Lemma *elimN*
Lemma *elimNf*
Lemma *elimTn*
Lemma *elimFn*

Cluster 183

Lemma *subnS*

Lemma addn_gt0
Lemma gtn_max
Lemma gtn_min
Lemma leq_mul2l
Lemma mul2n
Lemma doubleD
Lemma leq_double
Lemma leq_Sdouble
Lemma odd_double
Lemma size_rcons
Lemma belast_rcons
Lemma count_cat
Lemma all_pred0
Lemma rot_size_cat
Lemma has_filter
Lemma rot_uniq
Lemma index_cat
Lemma rot_rotr
Lemma rotr_rotr
Lemma cardUI
Lemma cardC
Lemma card2
Lemma disjoint_cat
Lemma card_unit
Lemma card_bool
Lemma val_enum_ord
Lemma size_enum_ord
Lemma index_enum_ord
Lemma bump_addl
Lemma inord_val

Cluster 184

Lemma maxn_minl
Lemma allP
Lemma allPn
Lemma leq_size_perm
Lemma rot_rot
Lemma subseq_uniqP
Lemma cardU1
Lemma cardD1
Lemma subset_cardP
Lemma or4P

Cluster 185

Lemma leqn0
Lemma lt0n
Lemma eqn0Ngt
Lemma addn_negb
Lemma eqb0
Lemma lt0b
Lemma sub1b
Lemma oddb
Lemma double_gt0
Lemma double_eq0
Lemma size_behead
Lemma nth_nil
Lemma set_nth_nil
Lemma drop0
Lemma take0
Lemma headI
Lemma mask0
Lemma mask1
Lemma mask_cons
Lemma has_mask_cons
Lemma mem_mask_cons
Lemma behead_map
Lemma andbN
Lemma andNb
Lemma orbN
Lemma orNb
Lemma implybF
Lemma addbT

Cluster 186

Lemma eqn_add2r
Lemma subnDr
Lemma ltnn
Lemma ltn_add2l
Lemma leq_add2r
Lemma leq_subLR
Lemma ltn_subRL
Lemma geq_max
Lemma geq_min
Lemma mulnSr
Lemma leq_mul2r
Lemma expnSr
Lemma expnAC
Lemma muln2
Lemma ltn_double

Lemma *ltn_Sdouble*
Lemma *doubleMl*
Lemma *doubleMr*
Lemma *mulnn*
Lemma *size_nseq*
Lemma *last_rcons*
Lemma *cat_rcons*
Lemma *rcons_cat*
Lemma *count_pred0*
Lemma *count_predT*
Lemma *has_pred0*
Lemma *has_predT*
Lemma *rev_cons*
Lemma *rev_cat*
Lemma *rev_rcons*
Lemma *mem_seq1*
Lemma *rcons_uniq*
Lemma *index_mem*
Lemma *rotr1_rcons*
Lemma *map_rcons*
Lemma *size_mkseq*
Lemma *card0*
Lemma *card1*
Lemma *properxx*
Lemma *subset_disjoint*
Lemma *disjointU1*
Lemma *size_image*
Lemma *mem_image*
Lemma *image_iinv*
Lemma *card_ord*
Lemma *leq_bump2*
Lemma *sub_ordK*

Cluster 187

Lemma *geq_minr*
Lemma *all_predT*
Lemma *index_size*
Lemma *ord_enum_uniq*
Lemma *sub_ord_proof*

Cluster 188

Lemma *neq_ltn*
Lemma *eqn_mul2l*
Lemma *eqn_mul2r*

Lemma *ltn_mul2l*
Lemma *ltn_mul2r*
Lemma *expn_eq0*
Lemma *has_rcons*
Lemma *all_rcons*
Lemma *rot0*
Lemma *size_rot*
Lemma *has_rot*
Lemma *rotr_size_cat*
Lemma *map_rot*
Lemma *cardX*

Cluster 189

Lemma *nat_irrelevance*
Lemma *subSnn*
Lemma *lt_irrelevance*
Lemma *ltn_add2r*
Lemma *maxnK*
Lemma *maxKn*
Lemma *minnK*
Lemma *minKn*
Lemma *mulSnr*
Lemma *sqrn_gt0*
Lemma *has_seq1*
Lemma *all_seq1*
Lemma *rot_size*
Lemma *perm_cons*
Lemma *enumT*
Lemma *enum0*
Lemma *disjoint_cons*
Lemma *size_codom*
Lemma *f_iinv*
Lemma *card_codom*
Lemma *canF_eq*
Lemma *card_sig*
Lemma *cast_ord_id*
Lemma *cast_ord_comp*
Lemma *nth_codom*
Lemma *bumpS*
Lemma *unbumpS*

Cluster 190

Lemma *leqif_geq*
Lemma *eq_find*

Lemma *eq_filter*
Lemma *eq_mkseq*
Lemma *eq_pick*
Lemma *eq_card*
Lemma *enum_rank_subproof*
Lemma *prop_congr*
Lemma *rwP2*

Cluster 191

Lemma *addn_eq0*
Lemma *mulnb*
Lemma *negb_and*
Lemma *negb_or*
Lemma *andbK*
Lemma *andKb*
Lemma *orbK*
Lemma *orKb*
Lemma *negb_imply*
Lemma *implybE*
Lemma *implyNb*
Lemma *implybN*
Lemma *implybNN*
Lemma *addbN*
Lemma *addNb*

Cluster 192

Lemma *ltn_sub2r*
Lemma *ltn_sub2l*
Lemma *leq_pmull*
Lemma *map_id_in*
Lemma *pmap_uniq*
Lemma *eq_disjoint0*
Lemma *eq_disjoint1*

Cluster 193

Lemma *expnD*
Lemma *nth_rcons*
Lemma *size_rev*
Lemma *mask_true*
Lemma *mem_pmap*
Lemma *flattenK*
Lemma *size_allpairs*

Cluster 194

Lemma eq_count
Lemma eq_has
Lemma eq_all
Lemma canLR_in
Lemma canRL_in
Lemma canLR_on
Lemma canRL_on
Lemma sub_in12
Lemma sub_in21

Cluster 195

Lemma ltn_neqAle
Lemma minnE
Lemma uniq_catC
Lemma card_sub
Lemma lift_max
Lemma card_sum

Cluster 196

Lemma sqrn_sub
Lemma uniq_size_uniq
Lemma rot_add_mod
Lemma count_enumP
Lemma predOPn
Lemma pcan_enumP

Cluster 197

Lemma leq_mul
Lemma sqrnD
Lemma sqrnD_sub
Lemma take_cat
Lemma filter_pred1_uniq
Lemma nth_uniq
Lemma rev_rot
Lemma map_subseq
Lemma card_preim
Lemma neq_bump

Cluster 198

Lemma *mem_take*
Lemma *eq_forallb*
Lemma *card_seq_sub*

Cluster 199

Lemma *uniq_enumP*
Lemma *eq_image*
Lemma *mono2W*
Lemma *monoLR*
Lemma *monoRL*
Lemma *mono2W_in*

Cluster 200

Lemma *fact_gt0*
Lemma *filter_subseq*

Cluster 201

Lemma *eq_proper*
Lemma *eq_proper_r*
Lemma *enum_ordS*
Lemma *rwP*

Cluster 202

Lemma *eqnP*
Lemma *all_predI*
Lemma *leq_bump*

Cluster 203

Lemma *subnS*
Lemma *addn_gt0*
Lemma *gtn_max*
Lemma *gtn_min*
Lemma *leq_mul2l*
Lemma *mul2n*
Lemma *doubleD*
Lemma *leq_double*
Lemma *leq_Sdouble*
Lemma *odd_double*
Lemma *cardUI*
Lemma *cardC*

Lemma *card2*
Lemma *disjoint_cat*
Lemma *preim_iinv*
Lemma *card_unit*
Lemma *card_bool*
Lemma *val_enum_ord*
Lemma *size_enum_ord*
Lemma *index_enum_ord*
Lemma *bump_addl*
Lemma *inord_val*
Lemma *card_prod*

Cluster 204

Lemma *subSKn*
Lemma *ltnNge*
Lemma *maxnSS*
Lemma *minnSS*
Lemma *ltn_sqr*
Lemma *leq_sqr*
Lemma *eqn_sqr*
Lemma *cardE*
Lemma *cardT*
Lemma *cardC1*
Lemma *subsetE*
Lemma *card_image*
Lemma *nth_ord_enum*

Cluster 205

Lemma *subn_if_gt*
Lemma *mulE*
Lemma *subxx_hint*

Cluster 206

Lemma *ltngtP*
Lemma *disjoint_has*
Lemma *val_ord_enum*
Lemma *sum_enum_uniq*

Cluster 207

Lemma *ltn_neqAle*
Lemma *leq_sub2r*

Lemma maxn_idPl
Lemma minnE
Lemma leq_exp2r
Lemma mem_seq_sub_enum
Lemma sub_enum_uniq
Lemma card_sub
Lemma nth_enum_ord
Lemma lift_max
Lemma card_sum

Cluster 208

Lemma sub0n
Lemma min0n
Lemma mul0n
Lemma invF_f
Lemma f_invF
Lemma cast_ordK
Lemma cast_ordKV
Lemma enum_valK

Cluster 209

Lemma neq_ltn
Lemma gtn_eqF
Lemma leq_total
Lemma subSn
Lemma minn_idPl
Lemma eqn_mul2l
Lemma eqn_mul2r
Lemma ltn_mul2l
Lemma ltn_mul2r
Lemma expn_eq0
Lemma negb_exists_in
Lemma nth_image
Lemma cardX

Cluster 210

Lemma leq_pexp2l
Lemma ltn_pexp2l
Lemma mem_sum_enum

Cluster 211

```
Lemma expnM
Lemma image_injP
Lemma seq_sub_pickleK
Lemma unbump_addl
```

Cluster 212

```
 Lemma subnn
Lemma bij_on_image
Lemma injF_bij
Lemma canF_sym
```

References

[1] J. Heras and E. Komendantskaya. Recycling Proof Patterns in Coq: Case Studies. *Journal Mathematics in Computer Science, accepted*, 2014.