A Formalization of Finite Group Theory: Part III

David M. Russinoff
david@russinoff.com

This is the third and final installment of an exposition of an ACL2 formalization of finite group theory. Part I covers groups and subgroups, cosets, normal subgroups, and quotient groups. Part II extends the theory in the developmnent of group homomorphisms and direct products, which are applied in a proof of the Fundamental Theorem of Finite Abelian Groups. The central topics of the present paper are the symmetric groups and the Sylow theorems, which pertain to subgroups of prime power order. Since these theorems are based on the conjugation of subgroups, an example of a group action on a set, their presentation is preceded by a comprehensive treatment of group actions. Our final result is mainly an application of the Sylow theorems: after showing that the alternating group of order 60 is simple (i.e., has no proper normal subgroup), we prove that no group of non-prime order less than 60 is simple. The combined content of the `groups` directory is a close approximation to that of an advanced undergraduate course taught by the author in 1976.

1 Introduction

This is the third and final installment of an exposition of an ACL2 formalization of finite group theory. Part I [1], which was presented at ACL2 2022, covers groups and subgroups, cosets, normal subgroups, and quotient groups. Part II [2], a companion paper in this workshop, extends the theory in the developmnent of group homomorphisms and direct products, which are applied in a proof of the Fundamental Theorem of Finite Abelian Groups. The present paper is an account of four books of the directory `projects/groups`—`symmetric`, `actions`, `sylow`, and `simple`—that have been appended to the seven books described in Parts I and II. Part I is a prerequisite for a reading of this paper. Parts II and III are largely independent, aside from several explicit references to Part II contained herein.

The central topics covered here are the symmetric groups `(sym n)` (Section 2), consisting of the permutations of the list `(0 1 2 ... n-1)`, and the Sylow theorems (Section 4), which pertain to subgroups of prime power order. Since these theorems are based on the conjugation of subgroups, an example of a group action on a set, their presentation is preceded by a comprehensive treatment of group actions (Section 3). Our final result (Section 5) is mainly an application of the Sylow theorems: after showing that the alternating group `(alt 5)`, of order 60, is simple (i.e., has no proper normal subgroup), we prove that no group of non-prime order less than 60 is simple.

2 Symmetric Groups

A *symmetric group* is the group of permutations of a given set under the operation of functional composition. The study of these groups has important applications in diverse areas of mathematics and physics, such as combinatorics, Galois theory, and quantum mechanics. They also provide a wide range of examples in group theory. Since the elements of the underlying set are irrelevant to the group structure, it is common to focus on the permutations of an initial segment of the positive integers, $\{1,2,\ldots,n\}$. In our ACL2 formalization, it is more natural to consider the list `(ninit n)` = `(0 1 ... n-1)` of the first `n` natural numbers.

Submitted to:
ACL2 Workshop 2023

2.1 Definition of (sym n)

In [2, Sec. 6], we define a permutation of an arbitrary list:

```
(defun permutationp (l m)
  (if (consp l)
      (and (member-equal (car l) m)
           (permutationp (cdr l) (remove1-equal (car l) m)))
    (endp m)))
```

In the special case of a list of distinct members, we have an equivalent formulation:

```
(defund permp (l m)
  (and (dlistp l) (dlistp m) (sublistp l m) (sublistp m l)))
(defthmd permp-permutationp
  (implies (and (dlistp l) (dlistp m))
           (iff (permutationp l m) (permp l m))))
```

The function perms recursively constructs a list of all permutations of a dlist. For a positive integer n, the element list of the symmetric group (sym n) is (slist n), the list of permutations of (ninit n):

```
(defund slist (n) (perms (ninit n)))
```

A permutation x in (slist n) may be viewed as a bijection of (ninit n), which maps an integer k to (nth k x). The group operation is functional composition:

```
(defun comp-perm-aux (x y l)
  (if (consp l)
      (cons (nth (nth (car l) y) x)
            (comp-perm-aux x y (cdr l)))
    ()))
(defund comp-perm (x y n)
  (comp-perm-aux x y (ninit n)))
```

The behavior of a product of permutations x and y is characterized by the following:

```
(defthm nth-comp-perm
  (implies (and (posp n) (natp k) (< k n))
           (equal (nth k (comp-perm x y n)) (nth (nth k y) x))))
```

More generally, we define the product of a list of permutations:

```
(defun comp-perm-list (l n)
  (if (consp l)
      (comp-perm (car l) (comp-perm-list (cdr l) n) n)
    (ninit n)))
```

The inverse operator is defined using the function index [2, Sec. 1], which gives the location of a member of a list:

```
(defun inv-perm-aux (x l)
  (if (consp l)
      (cons (index (car l) x) (inv-perm-aux x (cdr l)))
    ()))
(defund inv-perm (x n)
  (inv-perm-aux x (ninit n)))
```

It is easily shown that (slist n) is a dlist and that (car (slist n)) = (ninit n) is a left identity. After establishing closure, associativity, and the inverse property, we invoke the defgroup macro to construct the group:

```
(defgroup sym (n)
  (posp n)           ;parameter constraint
  (slist n)          ;element list
  (comp-perm x y n)  ;group operation
  (inv-perm x n))    'inverse operator
```

The length of `(slist n)` is easily computed:

```
(defthmd order-sym (implies (posp n) (equal (order (sym n)) (fact n))))
```

2.2 Transpositions

A *transposition* is a permutation in `(sym n)` that interchanges two indices and leaves all others fixed. The function `transpose` constructs the transposition of given indices `i` and `j`:

```
(defun transpose-aux (i j l)
  (if (consp l)
      (if (equal (car l) i)
          (cons j (transpose-aux i j (cdr l)))
        (if (equal (car l) j)
            (cons i (transpose-aux i j (cdr l)))
          (cons (car l) (transpose-aux i j (cdr l)))))
    ()))
(defund transpose (i j n) (transpose-aux i j (ninit n)))
```

We define a predicate that characterizes suitable arguments of `transpose`:

```
(defun trans-args-p (i j n)
  (and (posp n) (natp i) (natp j) (< i n) (< j n) (not (= i j))))
```

A transposition is a group element of ord 2:

```
(defthmd transpose-involution
  (implies (trans-args-p i j n)
           (equal (comp-perm (transpose i j n) (transpose i j n) n)
                  (ninit n))))
```

We need a predicate that recognizes a permutation as a transposition when the interchanged indices are unknown. First we define a function that identifies the least index that is not fixed by a given non-trivial permutation `p`:

```
(defun least-moved-aux (p k)
  (if (and (consp p) (equal (car p) k))
      (least-moved-aux (cdr p) (1+ k))
    k))
(defund least-moved (p) (least-moved-aux p 0))
```

The following predicate recognizes a transposition `p` in `(sym n)`:

```
(defund transp (p n)
  (let ((m (least-moved p)))
    (and (trans-args-p m (nth m p) n)
         (equal p (transpose m (nth m p) n)))))
(defthmd transp-transpose
  (implies (trans-args-p i j n)
           (transp (transpose i j n) n)))
```

A list of transpositions:

```
(defun trans-list-p (l n)
  (if (consp l)
      (and (transp (car l) n) (trans-list-p (cdr l) n))
    t))
```

We shall show that every element p of (sym n) is the product of a list (trans-list p n) of transpositions. Let m = (least-moved p), q = (transpose m (nth m p) n), and p$ = (comp-perm q p n). Note that (least-moved q) = m. Therefore, if k < m, then by nth-comp-perm,

```
(nth k p$) = (nth (nth k p) q) = (nth k q) = k,
```

while

```
(nth m p$) = (nth (nth m p) q) = m.
```

Thus, (least-moved p$) > m. This provides a measure for the following recursive definition:

```
(defun trans-list (p n)
  (declare (xargs :measure (nfix (- n (least-moved p)))))
  (let* ((m (least-moved p))
         (q (transpose m (nth m p) n))
           (p$ (comp-perm trans p n)))
    (if (and (posp n)
             (in p (sym n))
             (< m n))
        (cons trans (trans-list comp n))
      ())))
```

The desired result is easily proved using the induction scheme provided by the above definition:

```
(defthmd perm-prod-trans
  (implies (and (posp n) (in p (sym n)))
           (and (trans-list-p (trans-list p n) n)
                (equal (comp-perm-list (trans-list p n) n)
                       p))))
```

2.3 Parity

An element p of (sym n) may be represented in various ways as lists of transpositions, but we shall show that p determines whether the length of such a list is even or odd.

An *inversion* of p is a pair of indices (i . j) such that 0 ≤ i < j < n and (nth i p) > (nth j p). The *parity* of p is defined to be that of the number of its inversions. The formal definition is based on the list (pairs n) of pairs (i . j) such that 0 ≤ i < j < n. We extract from this list the list of inversions of p:

```
(defun invs-aux (p pairs)
  (if (consp pairs)
      (if (> (nth (caar pairs) p) (nth (cdar pairs) p))
          (cons (car pairs) (invs-aux p (cdr pairs)))
        (invs-aux p (cdr pairs)))
    ()))
(defund invs (p n) (invs-aux p (pairs n)))
```

We now define the parity of p:

```
(defund parity (p n) (mod (len (invs p n)) 2))
```

Accordingly, p is either *even* or *odd*:

```
(defund even-perm-p (p n) (equal (parity p n) 0))
(defund odd-perm-p (p n) (equal (parity p n) 1))
```

If p inverts i and j, then its inverse (inv-perm p n) inverts (nth j p) and (nth i p). It follows that p and (inv-perm p n) have the same number of inversions and therefore the same parity:

```
(defthmd parity-inv
  (implies (and (posp n) (in p (sym n)))
           (equal (parity (inv-perm p n) n)
                  (parity p n))))
```

The proof of the following formula for the parity of a product of permutations is more difficult and is omitted here (see the exposition in the comments in the proof script groups/symmetric.lisp):

```
(defthmd parity-comp-perm
  (implies (and (posp n) (in p (sym n)) (in r (sym n)))
           (equal (parity (comp-perm r p n) n)
                  (mod (+ (parity p n) (parity r n)) 2))))
```

It follows from parity-inv and parity-comp-perm that parity is preserved by conjugation:

```
(defthmd parity-conjugate
  (implies (and (posp n)
                (in p (sym n))
                (in a (sym n)))
           (equal (parity (conj p a (sym n)) n)
                  (parity p n))))
```

Note that a transposition of adjacent indices, (transpose i (1+ i) n), has exactly one inversion and is therefore odd, and every transposition is a conjugate of such a transposition:

```
(defthmd transpose-conjugate
  (implies (and (trans-args-p i j n) (< (1+ i) j))
           (equal (transpose i j n)
                  (comp-perm (transpose (1+ i) j n)
                             (comp-perm (transpose i (1+ i) n)
                                        (transpose (1+ i) j n)
                                        n)
                             n))))
```

We may conclude that every transposition is odd:

```
(defthmd transp-odd (implies (transp p n) (odd-perm-p p n)))
```

It follows that the parity of a product of a list of transpositions is that of the length of the list:

```
(defthmd parity-trans-list
  (implies (and (posp n) (trans-list-p l n))
           (equal (parity (comp-perm-list l n) n) (mod (len l) 2))))
```

In particular, this holds for the canonical factorization:

```
(defthmd parity-len-trans-list
  (implies (and (posp n) (in p (sym n)))
           (equal (parity p n) (mod (len (trans-list p n)) 2))))
```

2.4 Alternating Groups

The alternating group `(alt n)` is the subgroup of the symmetric group comprising the even permutations:

```
(defun even-perms-aux (l n)
  (if (consp l)
      (if (even-perm-p (car l) n)
          (cons (car l) (even-perms-aux (cdr l) n))
        (even-perms-aux (cdr l) n))
    ()))
(defund even-perms (n) (even-perms-aux (slist n) n))
(defsubgroup alt (n) (sym n)
  (posp n)
  (even-perms n))
```

It follows from `parity-conjugate` that `(alt n)` is a normal subgroup of `(sym n)`:

```
(defthmd alt-normal
  (implies (posp n) (normalp (alt n) (sym n))))
```

Assume `n` > 1 and let `s = (transpose 0 1 n)`, Then `s` is an odd permutation, and for any odd `p`, `(comp-perm s p n)` is even and therefore `p` belongs to `(lcoset s (alt n) (sym n))`. Thus, `(alt n)` has only two left cosets:

```
(defthmd subgroup-index-alt
  (implies (and (natp n) (> n 1))
           (equal (subgroup-index (alt n) (sym n)) 2)))
(defthmd order-alt
  (implies (and (natp n) (> n 1))
           (equal (order (alt n)) (/ (fact n) 2))))
```

3 Group Actions

3.1 Definition and the `defaction` Macro

Informally, an *action* of a group `g` on a dlist `d` is a mapping `a` that assigns to each element `x` of `g` and each member `s` of `d` a member `(act x s a g)` of `d`, satisfying two properties:

```
(act (e g) s a g) = s
(act x (act y s a g) a g) = (act (op x y g) s a g).
```

This may be viewed as a generalization of the operation of a group, which is an action of the group on its own element list. In fact, we use the same scheme for representing a group action as in the definition of a group: we define an action to be a matrix `a` of members of `d`, the first row of which is `d`, the *domain* of `a`:

```
(defmacro dom (a) `(car ,a))
```

The ith row of `a` defines the action of the ith element of `g` on the members of `d`:

```
(defund act (x s a g) (nth (ind s a) (nth (ind x g) a)))
```

where, according to the definition of `ind` [2, Sec. 1], `(ind s a) = (index s (dom a))`. Note that the first property above is automatic:

```
(act (e g) s a g) = (nth (ind s a) (nth 0 a)) = (nth (index s (dom a)) (dom a)) = a.
```

The defining predicate for an action calls the predicates `aclosedp` and `aassocp`, which exhaustively check the closure and associativity properties:

```
(defund actionp (a g)
  (and (groupp g)
       (dlistp (dom a))
       (consp (dom a))
       (matrixp a (order g) (len (dom a)))
       (aclosedp a g)
       (aassocp a g)))
```

It is easily verified that every group is indeed also an action:

```
(defthm actionp-groupp (implies (groupp g) (actionp g g)))
```

We have defined a macro for defining parametrized group actions, similar to `defgroup`:

`(defaction` *name args grp cond elts act*`)`,

where

- *grp* is the acting group;
- *cond* is a constraint that must be satisfied by the arguments *args*;
- *elts* is the domain;
- *act* is a term that specifies the action of a group element `x` on a member *s* of *elts*.

Conjugation is an important example. The form

```
(defaction conjugacy () g t (elts g) (conj s x g))
```

constructs the action `conjugacy` and proves three lemmas:

```
(DEFTHM ACTIONP-CONJUGACY
  (IMPLIES (GROUPP G) (ACTIONP (CONJUGACY G) G)))
(DEFTHM CONJUGACY-DOM
  (IMPLIES (GROUPP G)
           (EQUAL (DOM (CONJUGACY G)) (ELTS G))))
(DEFTHM CONJUGACY-ACT-REWRITE
  (IMPLIES (AND (GROUPP G) (IN X G) (IN S (CONJUGACY G)))
           (EQUAL (ACT X S (CONJUGACY G) G) (CONJ S X G))))
```

An action of a group g induces an action by any subgroup of g:

```
(defaction subaction (a g) h
  (and (actionp a g) (subgroupp h g))
  (dom a)
  (act x s a g))
```

As another example, if h is a subgroup of g, then we have an action of g on the left cosets of `h`, characterized by

```
(act x s (act-lcosets h g) g) = (lcoset (op x (car s) g) h g),
```

which is again constructed by `defaction`:

```
(defaction act-lcosets (h) g
  (subgroupp h g)
  (lcosets h g)
  (lcoset (op x (car s) g) h g))
```

3.2 Orbits and Stabilizers

If s is in the domain of an action a, the *orbit* of s is the ordered list of all r in (dom a) such that r = (act x s a g) for some x in g:

```
(defun orbit-aux (s a g l)
  (if (consp l)
      (let ((val (act (car l) s a g))
            (res (orbit-aux s a g (cdr l))))
        (if (member-equal val res)
            res
          (insert val res a)))
    ()))
(defund orbit (s a g) (orbit-aux s a g (elts g)))
```

We also define the list of all orbits of an action:

```
(defun orbits-aux (a g l)
  (if (consp l)
      (let ((res (orbits-aux a g (cdr l))))
        (if (member-list (car l) res)
            res
          (cons (orbit (car l) a g) res)))
    ()))
(defund orbits (a g) (orbits-aux a g (dom a)))
```

It is easily shown that every element of the domain belongs to its own orbit and that intersecting orbits are equal (i.e., distinct orbits are disjoint). It follows that appending the list of orbits yields a permutation of the domain:

```
(defthmd append-list-orbits
  (implies (actionp a g) (permp (append-list (orbits a g)) (dom a))))
```

Note that in the case of the conjugacy action, the orbit of a group element x is the conjugacy class (conjs x g) and the class equation [1, Sec. 8] is a special case of append-list-orbits.

The *stabilizer* of an element s of the domain of a is the ordered subgroup of g comprising all x such that (act x s a g) = s:

```
(defun stab-elts-aux (s a g l)
  (if (consp l)
      (if (equal (act (car l) s a g) s)
          (cons (car l) (stab-elts-aux s a g (cdr l)))
        (stab-elts-aux s a g (cdr l)))
    ()))
(defund stab-elts (s a g) (stab-elts-aux s a g (elts g)))
(defsubgroup stabilizer (s a) g
  (and (actionp a g) (member-equal s (dom a)))  ;constraints
  (stab-elts s a g))                            ;domain
```

If r is in the orbit of s, then for some x in g, (act x s a g) = r. The function actor is defined to produce such a value x. This gives rise to the following functions, which may be shown to be inverse bijections between (lcosets (stabilizer s a g) g) and (orbit s a g):

```
(defund lcosets2orbit (c s a g) (act (car c) s a g))
(defund orbit2lcosets (r s a g) (lcoset (actor r s a g) (stabilizer s a g) g))
```

Therefore, the lengths of these two lists are equal, and the following is a consequence of lagrange:

```
(defthmd stabilizer-orbit
  (implies (and (actionp a g) (in s a))
           (equal (* (order (stabilizer s a g)) (len (orbit s a g)))
                  (order g))))
```

Note that the centralizer of a group element x is its stabilizer under the conjugacy action, and the lemma len-conjs-cosets [1, Sec. 8] is a case of stabilizer-orbit.

3.3 Conjugation of Subgroups

The *conjugate* of a subgroup h of g by an element a of g is a subgroup comprising all conjugates of elements of h by a. We define this subgroup to have an ordered element list with respect to g, so that element lists of distinct conjugate subgroups cannot be permutations of one another:

```
(defun conj-sub-list-aux (l a g)
  (if (consp l)
      (insert (conj (car l) a g)
              (conj-sub-list-aux (cdr l) a g)
              g)
    ()))
(defund conj-sub-list (h a g) (conj-sub-list-aux (elts h) a g))
(defsubgroup conj-sub (h a) g
  (and (subgroupp h g) (in a g))
  (conj-sub-list h a g))
```

It is clear that a conjugate of h has the same order as h, and therefore if one conjugate is a subgroup of another, then they are equal. Since h itself need not be ordered with respect to g, h may not be a conjugate of itself, but the conjugate of h by (e g) (or by any element of h, for that matter) has the same elements as h:

```
(defthmd permp-conj-sub-e
  (implies (subgroupp h g)
           (permp (elts (conj-sub h (e g) g)) (elts h))))
```

Subgroup conjugation is an important example of a group action, the domain of which is a list of all conjugates of a given subgroup h of g:

```
(defun conjs-sub-aux (h g l)
  (if (consp l)
      (let ((c (conj-sub h (car l) g))
            (res (conjs-sub-aux h g (cdr l))))
        (if (member-equal c res)
            res
          (cons c res)))
    ()))
(defund conjs-sub (h g) (conjs-sub-aux h g (elts g)))
(defaction conj-sub-act (h) g (subgroupp h g) (conjs-sub h g) (conj-sub s x g))
```

We define the *normalizer* of a subgroup h of g to be the stabilizer of (conj-sub h (e g) g):

```
(defund normalizer (h g)
  (stabilizer (conj-sub h (e g) g)
              (conj-sub-act h g)
              g))
```

A subgroup is a normal subgroup of its normalizer:

```
(defthmd normalizer-normp
  (implies (subgroupp h g) (normalp h (normalizer h g))))
```

By `stabilizer-orbit`, the number of conjugates of `h` is the index of its normalizer, and therefore, `h` is normal in `g` iff `(mormalizer h g) = g`:

```
(defthmd index-normalizer
  (implies (subgroupp h g)
           (equal (len (conjs-sub h g))
                  (subgroup-index (normalizer h g) g))))
```

The normalizer of a conjugte of `h` is a conjugate of the normalizer of `h`:

```
(defthmd normalizer-conj-sub
  (implies (and (subgroupp m g)
                (member-equal c (conjs-sub m g)))
           (equal (normalizer c g)
                  (conj-sub (normalizer m g) (conjer-sub c m g) g))))
```

3.4 Induced Homomorphism into the Symmetric Group

An action `a` of a group `g` associates each element of `g` with a permutation of `(dom a)`. By identifying an element of `(dom a)` with its index in the list, we have an element of the symmetric group `(sym n)`, where `n = (len (dom a))`. If `x` is in `g` and `p` is the element of `(sym n)` corresponding to `x`, then for $0 \leq k < n$, the image of `k` under `p`, `(nth k p)`, is computed by the following:

```
(defund act-perm-val (x k a g)
  (index (act x (nth k (dom a)) a g)
         (dom a)))
```

Thus, the element of `(sym n)` corresponding to `x` may be computed recursively:

```
(defun act-perm-aux (x k a g)
  (if (zp k)
      ()
    (append (act-perm-aux x (1- k) a g)
            (list (act-perm-val x (1- k) a g)))))
(defund act-perm (x a g) (act-perm-aux x (order a) a g))

(defthmd act-perm-is-perm
  (implies (and (actionp a g) (in x g))
           (in (act-perm x a g) (sym (len (dom a))))))
(defthm act-perm-val-is-val
  (implies (and (actionp a g) (in x g) (member-equal k (ninit (order a))))
         (equal (nth k (act-perm x a g))
                (act-perm-val x k a g))))
```

It is clear that the identity of `g` corresponds to the identity of `(sym n)`, and that the group operation is preserved by this correspondence. Thus, we have a homomorphism from `g` into the symmetric group:

```
(defmap act-sym (a g)
  (elts g)
  (act-perm x a g))
(defthmd homomorphismp-act-sym
  (implies (actionp a g) (homomorphismp (act-sym a g) g (sym (order a)))))
```

The kernel of `(act-sym a g)` consists of the elements of `g` that act trivially on every element of `(dom a)`.

We have observed that every group `g` is an action of itself on its element list, with `(act g x s g) = (op x s g)`. The identity of `g` is the only element that acts trivially on every element (or, indeed, on any element). Therefore, every group `g` is isomorphic to a subgroup of `(sym (order g))`:

```
(defthm endomorphismp-act-sym-g
  (implies (groupp g) (endomorphismp (act-sym g g) g (sym (order g)))))
```

As another example, recall the action `act-lcosets` of a group `g` on the left cosets of a subgroup `h`, (Subsection 3.1). Clearly, the kernel of the homomorphism induced by this action is a subgroup of `h`:

```
(defthmd subgroup-kernel-act-cosets
  (implies (subgroupp h g)
           (subgroupp (kernel (act-sym (act-lcosets h g) g)
                              (sym (subgroup-index h g))
                              g)
                      h)))
```

This result has the following important consequence: If `p` is the least prime dividing the order of `g` and `h` is a subgroup of index `p`, then `h` is normal in `g`:

```
(defthmd index-least-divisor-normal
  (implies (and (subgroupp h g)
                (> (order g) 1)
                (equal (subgroup-index h g)
                       (least-prime-divisor (order g))))
           (normalp h g)))
```

The proof of this theorem also requires the observation that every homomorphism induces an endomorphism on the quotient of its kernel:

```
(defmap quotient-map (map g h)
  (lcosets (kernel map h g) g)
  (mapply map (car x)))
(defthmd endomorphismp-quotient-map
  (implies (homomorphismp map g h)
           (endomorphismp (quotient-map map g h) (quotient g (kernel map h g)) h)))
```

Tho prove `index-least-divisor-normal`, let

```
k = (kernel (act-sym (act-cosets h g) g) (sym (subgroup-index h g)) g).
```

We need only show that `k` and `h` have the same elements. (Since `h` need not be ordered with respect to `g`, the two subgroups may not be equal, but this will be sufficient to conclude that `h` is normal.) By `endomorphismp-quotient-map`, `(quotient g k)` is isomorphic to a subgroup of `(sym p)`, and therefore `(subgroup-index k g)` divides `(fact p)`, which implies `(subgroup-index k h)` divides `(fact (1- p))`. If `(subgroup-index k h)` > 1, then `(subgroup-index k g)` has a prime divisor `q`. Since `q` divides `(fact (1- p))`, `q` $<$ `p`. But since `q` divides `(order g)`, `q` $\geq$ `p` by assumption, a contradiction. Thus, `(subgroup-index k h)` = 1, which implies `(permp (elts k) (elts h))`.

4 Sylow Theorems

The Sylow theorems are a set of related results that provide information pertaining to the number of subgroups of prime power order of a finite group and the relations among them. These theorems form an important part of group theory, playing a critical role in the classification of finite groups.

Among these results is the statement that the order of a maximal p-subgroup of a finite group g is the maximal power of p that divides the order of g. As a first step, we shall prove that if h is a p-subgroup of g and p divides (subgroup-index p (normalizer h g)), then h is a proper subgroup of a larger p-subgroup of g, which may be constructed by first applying cauchy to construct a subgroup of (quotient (normalizer h g) h) of order p and then lifting it to g:

```
(defund extend-p-subgroup (h g p)
  (lift (cyclic (elt-of-ord p (quotient (normalizer h g) h))
                (quotient (normalizer h g) h))
        h
        (normalizer h g)))
(defthmd order-extend-p-subgroup
  (implies (and (subgroupp h g)
                (posp n)
                (elt-of-ord n (quotient (normalizer h g) h)))
           (let ((k (extend-p-subgroup h g n)))
             (and (subgroupp h k)
                  (subgroupp k g)
                  (equal (order k) (* n (order h)))))))
```

We recursively define a p-subgroup m = (sylow-subgroup g p) of g such that p does not divide the index of m in its normalizer:

```
(defun sylow-subgroup-aux (h g p)
  (declare (xargs :measure (nfix (- (order g) (order h)))))
  (if (and (subgroupp h g) (primep p)
           (divides p (subgroup-index h (normalizer h g))))
      (sylow-subgroup-aux (extend-p-subgroup h g p) g p)
    h))
(defund sylow-subgroup (g p) (sylow-subgroup-aux (trivial-subgroup g) g p))

(defthm index-sylow-subgroup
  (implies (and (groupp g) (primep p))
           (let ((m (sylow-subgroup g p)))
             (and (subgroupp m g)
                  (p-groupp m p)
                  (not (divides p (subgroup-index m (normalizer m g))))))))
```

We aim to show that p does not divide the index of m in g, i.e., (order m) is the maximal power of p that divides (order g). To this end, consider the action of g on the list of conjugates of m. This action has one orbit, the order of which is the index of the normalizer of m. We shall show that this index is congruent to 1 modulo p, and therefore not divisible by p.

Consider the restriction of this action to some p-subgroup h of g. Let c be a conjugate of m. By normalizer-conj-sub, the normalizer of c is a conjugate of the normalizer of m, and therefore the index of c in (normalizer c g) is not divisible by p.

Suppose x is an element of both h and (normalizer c g), but not an element of c. Since the order of x in g is a power of p, the order of the coset of x in (quotient (normalizer c g) c) is also a power of p, and p must divide (subgroup-index c (normalizer c g)), a contradiction. Thus, x is in h, then x is in (normalizer c g) iff x is in c.

By stabilizer-orbit, the length of the orbit of c under conjugation by h is 1 if h stabilizes c, and otherwise is divisible by p. But h stabilizes c iff h is a subgroup of (normalizer c g), and according to the above observation, this holds iff h is a subgroup of c:

```
(defthmd orbit-subaction-div-p
  (implies (and (subgroupp m g)
                (primep p)
                (p-groupp m p)
                (not (divides p (subgroup-index m (normalizer m g))))
                (subgroupp h g)
                (p-groupp h p)
                (in c (conj-sub-act m g)))
           (if (subgroupp h c)
               (equal (len (orbit c (subaction (conj-sub-act m g) g h) h)) 1)
             (divides p (len (orbit c (subaction (conj-sub-act m g) g h) h))))))
```

We first apply the above result to the case `h = m`. Since `m` is a subgroup of exactly 1 conjugate of `m`, there is exactly 1 orbit of length 1 and all others have length divisible by `p`:

```
(defthmd orbit-subaction-m-len-1
  (implies (and (subgroupp m g)
                (primep p)
                (p-groupp m p)
                (not (divides p (subgroup-index m (normalizer m g))))
                (in c (conj-sub-act m g)))
           (if (equal c (conj-sub m (e g) g))
               (equal (len (orbit c (subaction (conj-sub-act m g) g m) m)) 1)
             (divides p (len (orbit c (subaction (conj-sub-act m g) g m) m))))))
```

Appending all orbits yields the first Sylow theorem:

```
(defthmd sylow-1
  (implies (and (groupp g) (primep p))
           (let ((m (sylow-subgroup g p)))
             (equal (mod (len (conjs-sub m g)) p)
                    1))))
```

Since `(len (conjs-sub m g)) = (subgroup-index (normalizer m g) g)`, this length divides `(subgroup-index m g)`:

```
(defthmd sylow-2
  (implies (and (groupp g) (primep p))
           (let ((m (sylow-subgroup g p)))
             (divides (len (conjs-sub m g))
                      (subgroup-index m g)))))
```

Since `(len (conjs-sub m g)) = (subgroup-index (normalizer m g) g)` is not divisible by `p`, neither is `(subgroup-index m g)`:

```
(defthmd sylow-3
  (implies (and (groupp g) (primep p))
           (not (divides p (subgroup-index (sylow-subgroup g p) g)))))
```

The final Sylow theorem states that every `p`-subgroup of `g` is a subgroup of some conjugate of `m`. This is another consequence of `orbit-subaction-div-p`: If `h` were a counterexample to this claim, then according to `orbit-subaction-div-p`, the length of every orbit of `h` would be divisible by `p`, contradicting `mod-len-conjs-sub`.

The statement of the theorem requires the following finction, which searches a list `l` of subgroups of `g` for one that contains `h` as a subgroup:

```
(defun find-supergroup (h l)
  (if (consp l)
      (if (subgroupp h (car l))
          (car l)
        (find-supergroup h (cdr l)))
    ()))
(defthmd sylow-4
  (implies (and (groupp g) (primep p) (subgroupp h g) (p-groupp h p))
           (let* ((m (sylow-subgroup g p))
                  (k (find-supergroup h (conjs-sub m g))))
             (and (member-equal k (conjs-sub m g))
                  (subgroupp h k)))))
```

5 Simple Groups

A group is *simple* if has has no proper normal subgroup.

```
(defund proper-normalp (h g)
  (and (normalp h g) (> (order h) 1) (< (order h) (order g))))
```

Simple groups play an important role in the classification of finite groups. Since every group of prime order is simple, we focus on groups of composite order. One class of interest is that of the alternating groups. Note that `(alt 4)` is not simple, as may be verified by direct computation:

```
(defthmd alt-4-not-simple
  (proper-normalp (subgroup '((0 1 2 3) (1 0 3 2) (2 3 0 1) (3 2 1 0)) (sym 4))
                  (alt 4)))
```

However, `(alt n)` is simple for all $\texttt{n} \geq 5$. We shall prove this only for the case $\texttt{n} = 5$: `(alt 5)` is a simple group of order 60. In contrast to the more general theorem, our proof of this result is largely computational. We shall also prove, as an illustration of the Sylow theorems, that there are no simple groups of composite order less than 60.

5.1 Simplicity of `(alt 5)`

Let `h` be a normal subgroup of `g`. The function `conjs-list` [1, Sec. 8] constructs a list of the non-central conjugacy classes of `g`. We define `(select-conjs (conjs-list h) h)` to extract the conjugacy classes that are included in `h`:

```
(defun select-conjs (l h)
  (if (consp l)
      (if (in (caar l) h)
          (cons (car l) (select-conjs (cdr l) h))
        (select-conjs (cdr l) h))
    ()))
```

if we append the elements of that list together with the elements of `h` that belong to the center of `g`, we have a permutation of `(elts h)`. In the case of interest the center happens to be trivial. This gives us an expression for the order of `h`:

```
(defthmd len-select-conjs
  (implies (and (normalp h g) (equal (cent-elts g) (list (e g))))
           (equal (order h)
                  (1+ (len (append-list (select-conjs (conjs-list g) h)))))))
```

Thus, `(order h) = (1+ (len (append-list l)))` for some sublist `l` of `(conjs-list g)`. We need only compute this value for all such sublists of `(conjs-list (alt 5))` and observe that none of these values is a proper divisor of 60.

However, the function `conjs-list` is computationally impractical for a group of order 60. We define a more efficient and provably equivalent function, `conjs-list-fast`, based on a tail-recursive version of conjs. The lengths of the conjugacy classes of `(alt 5)` can be easily computed using this function:

```
(defun lens (l)
  (if (consp l)
      (cons (len (car l)) (lens (cdr l)))
    ()))
(defthmd lens-conjs-list-alt-5
  (equal (lens (conjs-list-fast (alt 5))) '(20 12 12 15)))
```

Clearly, no list of distinct members of this list has a sum that is a proper divisor of 60. Once we establish this simple fact (which requires some work), our theorem follows from `lagrange`:

```
(defthmd alt-5-simple (not (proper-normalp h (alt 5))))
```

5.2 Groups of Lesser Order

For every group `g` of composite `n` < 60, we shall construct a proper normal subgroup of `g`. We begin with the case of a prime power: `n = (expt p k)`, where `k` > 1. By `center-p-group` (book `cauchy`), `(order (center g))` > 1. If `(order (center g))` $<$ `(order g)`, then `(center (g))` is a proper normal subgroup. In the remaining case, `(center g) = g`, and hence `g` is abelian. Thus we need only show that `g` has a proper subgroup. But this follows from `cauchy`, which guarantees an element of order `p`. This leads to the following definition and lemma:

```
(defund normal-subgroup-prime-power (p k g)
  (declare (ignore k))
  (if (< (order (center g)) (order g))
      (center g)
    (cyclic (elt-of-ord p g) g)))
(defthm proper-normalp-prime-power
  (implies (and (groupp g)
                (equal (order g) (expt p k))
                (primep p)
                (natp k)
                (> k 1))
           (proper-normalp (normal-subgroup-prime-power p k g) g)))
```

The rest of the proof is based mainly on the Sylow theorems. We consider various cases according to the prime factorization of `n`. As a notational convenience, we shall denote `(sylow-subgroup g p)` by `hp` and `(len (conjs-sub hp g))` by `np`.

Suppose `n = (* p q)`, where `p` and `q` are primes and `p` $<$ `q`. By the Sylow theorems, `nq` divides `p` and `(mod nq p) = 1`. It follows that `np = 1`, which implies `(sylow-subgroup g q)` is normal in `g`.

```
(defund normal-subgroup-pq (p q g)
  (declare (ignore p))
  (sylow-subgroup g q))
(defthm proper-normalp-pq
  (implies (and (groupp g) (equal (order g) (* p q))
                (primep p) (primep q) (< p q))
           (proper-normalp (normal-subgroup-pq p q g) g)))
```

Next, we consider the case `n = (* p p q)`, where `p` and `q` are primes. We must show that either `np = 1` or `nq = 1`. Suppose not. Since `np` divides `q` and `(mod np p) = 1`, $q > p$. Since `nq` divides `(* p p)` and `(mod nq q) = 1`, `nq = (* p p)` and `q` divides `(1- (* p p))`. Thus, `q` divides either `(1- p)` or `(1+ p)`. Since $q > p$, `q = (1+ p)`, which implies `p = 2`, `p = 3`, and `n = 12`. Since `n3 = 4` and each 3-Sylow subgroup has 2 non-trivial elements, `g` has 8 elements of order 3. Since `n2` >1, `g` has more than 4 elements of order dividing 4, a contradiction.

```
(defund normal-subgroup-ppq (p q g)
  (if (normalp (sylow-subgroup g p) g)
      (sylow-subgroup g p)
    (sylow-subgroup g q)))
(defthm proper-normalp-ppq
  (implies (and (groupp g)
                (equal (order g) (* p p q))
                (primep p)
                (primep q)
                (not (equal p q)))
           (proper-normalp (normal-subgroup-ppq p q g) g)))
```

There are eight remaining cases, which are treated individually: 24, 30, 36, 40, 42, 48, 54, and 56. Consider the case `n = 24`. Assume `n2` >1 and let `h21` and `h22` be distinct members of `(conj-subs h2 g)`. Then `(order h21) = order h22) = 8`. Let `k = (group-intersection h21 h22 g)`. Then `(order k)` ≤ 4 and by `len-products` [2, Sec. 3],

`(len (products h21 h22 g)) = (/ (* (order h1) (order h2)) (order k))` ≤ 24,

which implies `(order k) = 4` and `(len (products h21 h22 g)) = 16`. By `index-least-divisor-normal` (Subsection 3.4), `k` is normal in both `h21` and `h22`. It follows that `(normalizer k g)` contains `(products h21 h22 g)`. Consequently, `(order (normalizer k g))` ≥ 16, which implies `(normalizer k g) = g` and `k` is normal in `g`. Thus, we have the following:

```
(defund normal-subgroup-24 (g)
  (let* ((h2 (sylow-subgroup g 2))
         (h21 (car (conjs-sub h2 g)))
         (h22 (cadr (conjs-sub h2 g)))
         (k (group-intersection h21 h22 g)))
    (if (normalp h2 g)
        h2
      k)))
(defthm proper-normalp-24
  (implies (and (groupp g) (equal (order g) 24))
           (proper-normalp (normal-subgroup-24 g) g)))
```

We omit the other seven cases, which use the same techniques as illustrated as above. We combine these results in a function that splits into cases corresponding to the composite integers less than 60:

```
(defund normal-subgroup (g)
  (case (order g)
    (4 (normal-subgroup-prime-power 2 2 g))
    (6 (normal-subgroup-pq 2 3 g))
    (8 (normal-subgroup-prime-power 2 3 g))
    (9 (normal-subgroup-prime-power 3 2 g))
    (10 (normal-subgroup-pq 2 5 g))
    (12 (normal-subgroup-ppq 2 3 g))
    ...
    (56 (normal-subgroup-56 g))
```

```
    (57 (normal-subgroup-pq 3 19 g))
    (58 (normal-subgroup-pq 2 29 g))))

(defthm no-simple-group-of-composite-order<60
  (implies (and (natp n) (> n 1) (< n 60) (not (primep n))
                (groupp g) (equal (order g) n))
           (proper-normalp (normal-subgroup g) g)))
```

6 Conclusion

Our survey of this rich topic is far from complete. We anticipate enhancements of the theory, such as the representation of a permutation as a product of disjoint cycles, which is required, for example, for a general proof of the simplicity of `(alt n)`. However, the intended scope of the project has essentially been realized. The combined content of the `groups` directory is a close approximation to that of an advanced undergraduate course that the author taught at The Cooper Union in the Spring of 1976.

The concluding section of Part I discusses the long-term objective of a formalization of algebraic number theory. The next steps in this direction are elementary linear algebra and Galois theory, the first of which is underway. We note one important difference between our approaches to groups, on one hand, and fields and vector spaces on the other. As we have observed, our interest in finite groups and the importance of proof by induction on the order of a group led us away from the characterization of a group by means of encapsulated constrained functions in favor of an explicit defining predicate. On the other hand, since we are interested in both infinite and finite fields (and the role of induction is less critical even in the latter case), we are instead pursuing the encapsulation approach in the formalization of fields as well as finite dimensional vector spaces.. A progress report may be expected at the next ACL2 workshop.

References

[1] David M. Russinoff (2022): *A Formalization of Finite Froup Theory*. In: *ACL2 2022: 17th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas.

[2] David M. Russinoff (2023): *A Formalization of Finite Froup Theory: Part II*. In: *ACL2 2023: 18th International Workshop on the ACL2 Theorem Prover and its Applications*, Austin, Texas.