A Formalization of Finite Group Theory

David M. Russinoff
david@russinoff.com

Previous formulations of group theory in ACL2 and Nqthm, based on either `encapsulate` or `defn-sk`, have been limited by their failure to provide a path to proof by induction on the order of a group, which is required for most interesting results in this domain beyond Lagrange's Theorem (asserting the divisibility of the order of a group by that of a subgroup). We describe an alternative approach to finite group theory that remedies this deficiency, based on an explicit representation of a group as an operation table. We define a `defgroup` macro for generating parametrized families of groups, which we apply to the additive and multiplicative groups of integers modulo *n*, the symmetric groups, arbitrary quotient groups, and cyclic subgroups. In addition to a proof of Lagrange's Theorem, we provide an inductive proof of a theorem of Cauchy: *If the order of a group G is divisible by a prime p, then G has an element of order p.*

1 Introduction

Since ACL2 provides only limited support for quantification, modeling group theory in its logic is a challenging problem. A 1990 paper of Yuan Yu [7] presents a formal development of finite group theory in Nqthm based on the `defn-sk` macro (surviving in ACL2 as `defun-sk`), which he uses to define a predicate that characterizes a list whose members satisfy the group axioms with respect to a fixed operation. He also defines the notion of group homomorphism and proves that the kernel of any homomorphism is a normal subgroup, but this requires an additional `defn-sk` form in order to introduce a group with a different operation. The culmination of Yu's work is Lagrange's Theorem: *The order of a finite group is divisible by the order of any subgroup.* Any significant further development of the theory would require induction on the order of a group, which seems to be inaccessible through this method.

We are unaware of any ACL2 results in this domain that duplicate Yu's achievement. A similar approach based on `encapsulate` has been suggested [6], but while this provides a generalization to infinite groups, it otherwise shares the same limitations as the `defun-sk` method. Heras et al. [3] describe "a guideline to develop tools that simplify the formalizations related to algebraic structures in ACL2", but do not mention any results that are directly relevant to the theory of groups.

We shall present an ACL2 formalization of finite groups that provides for inductive proofs as well as computations on concrete groups. Our scheme is based on the definition of a group as an explicit operation table, i.e., a matrix of group elements. We define a `defgroup` macro that provides definitions of parametrized families of groups, which we apply to the additive and multiplicative groups of integers modulo *n*, the symmetric and alternating groups, arbitrary quotient groups, and cyclic subgroups. Our proof of Lagrange's theorem shares some features with Yu's, but we prove a stronger version stating that the order of a group is the product of that of a subgroup and its index. This leads naturally into an analysis of quotient groups, which lays the groundwork for a theorem of Cauchy: *If the order of a group G is divisible by a prime p, then G has an element of order p.* We present an inductive proof of this result, which illustrates the effectuality of our scheme. The proof, which resides in `books/workshops/2022/russinoff-groups/`, uses several number-theoretic results from `books/projects/quadratic-reciprocity/euclid.lisp`, including the following basic theorem of Euclid: *If a product of integers ab is divisible by a prime p, then p divides either a or b.*

2 Groups and Subgroups

In our formalization, a group is a square matrix, the first row of which is a list of the group elements:

```
(defmacro elts (g) `(car ,g))
(defmacro in (x g) `(member-equal ,x (elts ,g)))
(defmacro order (g) `(len (elts ,g)))
```

The *index* of a group element is its position in the list `(elts g)`:

```
(defun index (x l)
  (if (consp l)
      (if (equal x (car l))
          0
        (1+ (index x (cdr l))))
    ()))
(defmacro ind (x g) `(index x (elts g)))
```

The group operation is defined as a table access:

```
(defun op (x y g) (nth (ind y g) (nth (ind x g) g)))
```

We also define

```
(defmacro e (g) `(car (elts ,g)))
```

Note that `(nth (ind (e g) g) g)` = `(nth 0 g)` = `(elts g)` and therefore,

`(op (e g) y g)` = `(nth (ind y g) (elts g))` = `y`

i.e., `(e g)` is a left identity:

```
(defthm group-left-identity
  (implies (in x g) (equal (op (e g) x g) x)))
```

The left inverse `(inv x g)`, if it exists, is the group element of least index satisfying

`(op (inv x g) x g)` = `x`

defined as follows:

```
(defun inv-aux (x l g)
  (if (consp l)
      (if (equal (op (car l) x g) (e g))
          (car l)
        (inv-aux x (cdr l) g))
    ()))
(defun inv (x g) (inv-aux x (elts g) g))
```

The definition of a group is based on a set of predicates, including those representing the group axioms:

```
(defund groupp (g)
  (and (posp (order g))
       (matrixp g (order g) (order g))
       (dlistp (elts g))
       (not (in () g))
       (closedp g)
       (assocp g)
       (inversesp g)))
```

The predicate `matrixp` is a straightforward characterization of a matrix of given dimensions, and `dlistp` recognizes lists of distinct members. The condition that `NIL` is not a group element is unnecessary but avoids certain technical difficulties. The definition of `closedp` recursively searches for a a pair of group elements `x` and `y` such that `(op x y g)` is not in the group. This allows us to prove the theorem

```
(defthm group-closure
  (implies (and (groupp g) (in x g) (in y g))
           (in (op x y g) g)))
```

and also provides a counterexample if the search succeeds:

```
(defthm not-closedp-cex
  (implies (and (dlistp (elts g)) (not (closedp g)))
           (let* ((cex (closedp-cex g)) (x (car cex)) (y (cadr cex)))
             (and (in x g) (in y g)
                  (not (in (op x y g) g))))))
```

The latter result is useful in verifying `(closedp g)` for a conjectured group g. An analogous pair of theorems is derived for `assocp`, which searches for a triple that violates associativity, and `inversesp`, which searches for an element without a left inverse. Other basic properties, e.g., right identity, right inverse, and cancellation laws, follow easily.

Similarly, subgroups are defined by the predicate

```
(defun subgroupp (h g)
  (and (groupp g)
       (groupp h)
       (sublistp (elts h) (elts g))
       (not (subgroupp-cex h g))))
```

where `subgroupp-cex` exhaustively searches for a pair of elements of `h` on which the two group operations disagree. Thus,

```
(defthm subgroup-op
  (implies (and (subgroupp h g) (in x h) (in y h))
           (equal (op x y h) (op x y g))))
```

Again, the search produces a counter-example if it exists:

```
(defthm not-subgroupp-cex
  (implies (and (groupp g) (groupp h)
                (sublistp (elts h) (elts g))
                (not (subgroupp h g)))
           (let* ((cex (subgroupp-cex h g)) (x (car cex)) (y (cadr cex)))
             (and (in x h) (in y h)
                  (not (equal (op x y h) (op x y g)))))))
```

The following results are also readily derived from the definition:

```
(defthm subgroup-e
  (implies (subgroupp h g) (equal (e h) (e g))))

(defthm subgroup-inv
  (implies (and (subgroupp h g) (in x h))
           (equal (inv x h) (inv x g))))
```

We also define a function `subgroup`, which constructs the subgroup of a given group with a given element list if such a group exists. An illustration will be provided in the next section.

3 Parametrized Groups

The macro `defgroup` is based on the following encapsulation, which constrains three functions representing the list of elements of a group, the group operation, and its inverse operator:

```
(encapsulate (((glist) => *) ((gop * *) => *) ((ginv *) => *))
  (local (defun glist () (list 0)))
  (local (defun gop (x y) (+ x y)))
  (local (defun ginv (x) x))
  (defthm consp-glist (consp (glist)))
  (defthm dlistp-glist (dlistp (glist)))
  (defthm g-non-nil (not (member-equal () (glist))))
  (defthm g-identity
    (implies (member-equal x (glist))
             (equal (gop (car (glist)) x) x)))
  (defthm g-closed
    (implies (and (member-equal x (glist))
                  (member-equal y (glist)))
             (member-equal (gop x y) (glist))))
  (defthm g-assoc
    (implies (and (member-equal x (glist))
                  (member-equal y (glist))
                  (member-equal z (glist)))
             (equal (gop x (gop y z)) (gop (gop x y) z))))
  (defthm g-inverse
    (implies (member-equal x (glist))
             (and (member-equal (ginv x) (glist))
                  (equal (gop (ginv x) x) (car (glist)))))))
```

Our definition of the group `(g)` is based on the constrained functions gop and `glist`:

```
(defun g-row (x m)
  (if (consp m)
      (cons (gop x (car m)) (g-row x (cdr m)))
    ()))
(defun g-aux (l m)
  (if (consp l)
      (cons (g-row (car l) m) (g-aux (cdr l) m))
    ()))
(defun g () (let ((l (glist))) (g-aux l l)))
```

Using the results discussed in Section 2, we prove the theorem

```
(defthm groupp-g (groupp (g)))
```

along with three results characterizing the group structure:

```
(defthm glist-elts (equal (elts (g)) (glist)))
(defthm op-g-rewrite
  (implies (and (in x (g)) (in y (g)))
           (equal (op x y (g)) (gop x y))))
(defthmd inv-g-rewrite (implies (in x (g)) (equal (inv x (g)) (ginv x))))
```

The macro defines a parametrized family of groups given a parameter list, a predicate that the parameters must satisfy, and three terms corresponding to the above constrained functions. For example, the additive group of integers modulo `n` is generated by the following:

```
(defun ninit (n) (if (zp n) () (append (ninit (1- n)) (list (1- n)))))
(defun z+-op (x y n) (mod (+ x y) n))
(defun z+-inv (x n) (mod (- x) n))
(defgroup z+ (n) (posp n) (ninit n) (z+-op x y) (z+-inv x))
```

Prior to the evaluation of a `defgroup` form, a set of preliminary rewrite rules corresponding to the seven exported theorems of the encapsulation must be proved. The first three, which state that the specified list of elements is a non-`NIL` list of distinct non-`NIL` members, are generally trivial. In this case, the remaining four are also easy to prove:

```
(defthm z+-identity
  (implies (and (posp n) (member-equal x (ninit n)))
           (equal (z+-op 0 x n) x)))
(defthm z+-closed
  (implies (and (posp n)
                (member-equal x (ninit n))
                (member-equal y (ninit n)))
           (member-equal (z+-op x y n) (ninit n))))
(defthm z+-assoc
  (implies (and (posp n)
                (member-equal x (ninit n))
                (member-equal y (ninit n))
                (member-equal z (ninit n)))
           (equal (z+-op x (z+-op y z n) n)
                  (z+-op (z+-op x y n) z n))))
(defthm z+-inverse
  (implies (and (posp n) (member-equal x (ninit n)))
           (and (member-equal (z+-inv x n) (ninit n))
                (equal (z+-op (z+-inv x n) x n) 0))))
```

The family `(z+ n)` is then defined by the above `defgroup` form, which also automatically proves four theorems:

```
(DEFTHM GROUPP-Z+ (IMPLIES (POSP N) (GROUPP (Z+ N))))
(DEFTHM Z+-ELTS (IMPLIES (POSP N) (EQUAL (ELTS (Z+ N)) (NINIT N))))
(DEFTHM Z+-OP-REWRITE
  (IMPLIES (AND (POSP N) (IN X (Z+ N)) (IN Y (Z+ N)))
           (EQUAL (OP X Y (Z+ N)) (Z+-OP X Y N))))
(DEFTHM Z+-INV-REWRITE
  (IMPLIES (AND (POSP N) (IN X (Z+ N)))
           (EQUAL (INV X (Z+ N)) (Z+-INV X N))))
```

Each of these results is derived by the same functional instantiation of the corresponding lemma pertaining to the constrained constant g. For example,

```
(DEFTHM GROUPP-Z+ (IMPLIES (POSP N) (GROUPP (Z+ N)))
  :HINTS (("Goal"
           :USE ((:FUNCTIONAL-INSTANCE GROUPP-G
                  (GLIST (LAMBDA NIL (IF (POSP N) (NINIT N) (GLIST))))
                  (GOP (LAMBDA (X Y) (IF (POSP N) (Z+-OP X Y) (GOP X Y))))
                  (GINV (LAMBDA (X) (IF (POSP N) (Z+-INV X) (GINV X))))
                  (G-ROW (LAMBDA (X M) (IF (POSP N) (Z+-ROW X M N) (G-ROW X M))))
                  (G-AUX (LAMBDA (L M) (IF (POSP N) (Z+-AUX L M N) (G-AUX L M))))
                  (G (LAMBDA NIL (IF (POSP N) (Z+ N) (G)))))))))
```

Note that `Z+-ROW` and `Z+-AUX` are auxiliary functions that are generated by `defgroup` along with `Z+`.

The multiplicative group `(z* n)` of integers modulo `n` is similarly generated, replacing addition with multiplication:

```
(defun z*-op (x y n) (mod (* x y) n))
```

where we assume `(and (natp n) (> n 1))`. The element list is the sublist `(rel-primes n)` of `(ninit n)` consisting of integers relatively prime to `n`. This list is computed using the greatest common divisor function, `g-c-d`, which is treated in `euclid.lisp`. It is clear that `(car (rel-primes n))` = 1 satisfies the identity property. Closure and associativity follow from the established properties of `g-c-d` and `mod`. For the definition of the inverse operator, we appeal to the following property of `g-c-d`:

```
(defthm g-c-d-linear-combination
  (implies (and (integerp x) (integerp y))
           (= (+ (* (r-int x y) x) (* (s-int x y) y))
              (g-c-d x y))))
```

Thus, `(g-c-d x y)` is a linear combination of `x` and `y` with integer coefficients `(r-int x y)` and `(s-int x y)`. We define

```
(defun z*-inv (x n) (mod (r-int x n) n))
```

and the required property follows from `g-c-d-linear-combination`:

```
(defthm z*-inverse
  (implies (and (natp n) (> n 1) (member-equal x (rel-primes n)))
           (and (member-equal (z*-inv x n) (rel-primes n))
                (equal (z*-op (z*-inv x n) x n) 1))))
```

Thus, we have

```
(defgroup z* (n) (and (natp n) (> n 1)) (rel-primes n) (z*-op x y n) (z*-inv x n))
```

and the usual four generated theorems, including

```
(DEFTHM GROUPP-Z* (IMPLIES (AND (NATP N) (> N 1)) (GROUPP (Z* N))))
```

For example, evaluation of `(z* 15)` yields a group or order 8:

```
((1 2 4 7 8 11 13 14)
 (2 4 8 14 1 7 11 13)
 (4 8 1 13 2 14 7 11)
 (7 14 13 4 11 2 1 8)
 (8 1 2 11 4 13 14 7)
 (11 7 14 2 13 1 8 4)
 (13 11 7 1 14 8 4 2)
 (14 13 11 8 7 4 2 1))
```

As an illustration of the `subgroup` function mentioned at the end of Section 2, we observe that

```
(subgroup '(1 4 7 13) (z* 15))
```

is

```
((1 4 7 13)
 (4 1 13 7)
 (7 13 4 1)
 (13 7 1 4))
```

and `(subgroupp (subgroup '(1 4 7 13) (z* 15)) (z* 15))` = T. Note that in order for `sub-group` to succeed in generating a group, the first member of the supplied list must be the identity element.

The element list of the symmetric group `(sym n)` is given by

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

where `(perms l)` returns a list of all permutations of a list `l`. The group operation is composition, defined by

```
(defun comp-perm-aux (p r l)
  (if (consp l)
      (cons (nth (nth (car l) r) p) (comp-perm-aux p r (cdr l)))
    ()))
(defun comp-perm (p r n) (comp-perm-aux p r (ninit n)))
```

and the inverse operator is

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

Once we establish the required preliminary lemmas (which has not yet been done at the time of writing), we shall invoke

```
(defgroup sym (n) (posp n) (slist n) (comp-perm x y n) (inv-perm x n)).
```

In the meantime, we have defined a weaker version of `defgroup` that defines a family of groups without proving any theorems, and does not require either the parameter constraint or the inverse operator:

```
(defgroup-light sym (n) (slist n) (comp-perm x y n))
```

This allows us to analyze concrete groups of the family. For example, `(sym 3)` is a group of order 6,

```
(((0 1 2) (0 2 1) (1 0 2) (1 2 0) (2 0 1) (2 1 0))
 ((0 2 1) (0 1 2) (2 0 1) (2 1 0) (1 0 2) (1 2 0))
 ((1 0 2) (1 2 0) (0 1 2) (0 2 1) (2 1 0) (2 0 1))
 ((1 2 0) (1 0 2) (2 1 0) (2 0 1) (0 1 2) (0 2 1))
 ((2 0 1) (2 1 0) (0 2 1) (0 1 2) (1 2 0) (1 0 2))
 ((2 1 0) (2 0 1) (1 2 0) (1 0 2) (0 2 1) (0 1 2)))
```

and we can prove

```
(defthm groupp-sym-3 (groupp (sym 3)))
```

by direct computation. Note that the identity element of `(sym n)` is the trivial permutation `(ninit n)`.

To construct the alternating groups, we define a function `cyc` that converts an element of `(sym n)` to an alternative representation as a product of cycles. For example, in `(sym 5)`,

`(cyc '(2 3 4 1 0))` = `((0 2 4) (1 3))`,

We can derive the parity of a permutation from the observation that a cycle of odd (resp., even) length is a product of an even (resp., odd) number of transpositions. Thus, we define an even permutation as follows:

```
(defun even-cyc (cyc)
  (if (consp cyc)
      (if (evenp (len (car cyc)))
          (not (even-cyc (cdr cyc)))
        (even-cyc (cdr cyc)))
    t))
(defun even-perm (perm) (even-cyc (cyc perm)))
```

The function `even-perms` extracts the sublist of even permutations from a list. The alternating group `(alt n)` is the subgroup of `(sym n)` consisting of the even permutations:

```
(defgroup-light alt (n)
  (even-perms (slist n))
  (comp-perm x y n))
```

For example, `(alt 3)` is the group

```
(((0 1 2) (1 2 0) (2 0 1))
 ((1 2 0) (2 0 1) (0 1 2))
 ((2 0 1) (0 1 2) (1 2 0)))
```

and we can prove theorems such as `(subgroupp (alt 5) (sym 5))`.

4 Cosets and Lagrange's Theorem

For our purposes, we need only consider left cosets. Given an element `x` and a subgroup `h` of a group `g`, `(lcoset x h g)` is defined to be a list of all elements of `g` of the form `(op x y g)` that satisfy `(in y h)`. In particular, `(lcoset (e g) h g)` is a permutation of `(elts h)`. Our definition of `lcoset` ensures that this list is ordered by indices with respect to `g`. It follows that its members are distinct:

```
(defthm dlistp-lcosets
  (implies (and (subgroupp h g) (in x g))
           (dlistp (lcoset x h g)))))
```

It is also easily shown that the length of each coset is the order of the subgroup:

```
(defthm len-lcoset
  (implies (and (subgroupp h g) (in x g))
           (equal (len (lcoset x h g)) (order h))))
```

The following is a useful criterion for coset membership:

```
(defthmd member-lcoset-iff
  (implies (and (subgroupp h g) (in x g) (in y g))
           (iff (member-equal y (lcoset x h g))
                (in (op (inv x g) y g) h))))
```

As a consequence of this result, intersecting cosets have the same members, and the following may be derived from the ordering property:

```
(defthmd equal-lcoset
  (implies (and (subgroupp h g) (in x g) (in y g)
                (member-equal y (lcoset x h g)))
           (equal (lcoset y h g) (lcoset x h g))))
```

The list `(lcosets h g)` is constructed by traversing `(elts g)` and adding a coset to the list whenever an element is encountered that does not already appear in the list. By definition, the length of the list is the index of `h` in `g`:

```
(defun subgroup-index (h g) (len (lcosets h g)))
```

Our proof of Lagrange's Theorem is based on the list `(append-list (lcosets h g))`, produced by appending all members of `(lcosets h g)`. The above results lead to the following properties of this list:

```
(defthm dlistp-append-list-lcosets
  (implies (subgroupp h g)
           (dlistp (append-list (lcosets h g)))))
(defthm len-lcosets
  (implies (subgroupp h g)
           (equal (len (append-list (lcosets h g)))
                  (* (order h) (subgroup-index h g)))))
```

The proof of Lagrange's Theorem depends on the observation that if each of two lists of distinct members is a sublist of the other, then the lists have the same length:

```
(defthmd sublistp-equal-len
  (implies (and (dlistp l)
                (dlistp m)
                (sublistp l m)
                (sublistp m l))
           (equal (len l) (len m))))
```

The hypotheses of this lemma are readily established for the lists `(append-list (lcosets h g))` and `(elts g)`, and the theorem follows:

```
(defthm lagrange
  (implies (and (groupp g) (subgroupp h g))
           (equal (* (order h) (subgroup-index h g))
                  (order g))))
```

5 Normal Subgroups and Quotient Groups

For elements x and y of a group g, the conjugate of x by y is defined by

```
(defund conj (x y g) (op (op (inv y g) x g) y g))
```

Note that this computation returns x iff x and y commute.

A normal subgroup h of g is recognized by the predicate (normalp h g), which first requires that h be a subgroup of g and then exhaustively checks that every conjugate of every element of h is an element of h. As usual, we have the following two results:

```
(defthm normalp-conj
  (implies (and (normalp h g) (in x h) (in y g))
           (in (conj x y g) h)))
(defthmd not-normalp-cex
  (let* ((cex (normalp-cex h g)) (x (car cex)) (y (cadr cex)))
    (implies (and (subgroupp h g) (not (normalp h g)))
             (and (in x h) (in y g) (not (in (conj x y g) h))))))
```

We shall apply defgroup to define the group (quotient g h) when h is a normal subgroup of g. The elements of this group are the members of (lcosets h g), and the identity element is the coset of (e g):

```
(defun qe (h g) (lcoset (e g) h g))
```

Thus, we must rearrange (lcosets h g), moving this element to the front of the list:

```
(defun qlist (h g) (cons (qe h g) (remove1-equal (qe h g) (lcosets h g))))
```

The group operation is

```
(defun qop (x y h g) (lcoset (op (car x) (car y) g) h g))
```

and the inverse operator is

```
(defun qinv (x h g) (lcoset (inv (car x) g) h g))
```

The closure property is trivial. The remaining properties required by defgroup (identity, associativity, and inverse) may be derived from the following result, which is a consequence of normalp-conj and member-lcoset-iff:

```
(defthm op-qop
  (implies (and (normalp h g)
                (member-equal x (qlist h g)) (member-equal y (qlist h g))
                (member-equal a x) (member-equal b y))
           (member-equal (op a b g) (qop x y h g))))
```

We may now invoke

```
(defgroup quotient (g h) (normalp h g) (qlist h g) (qop x y h g) (qinv x h g))
```

which generates the usual four results, including

```
(DEFTHM GROUPP-QUOTIENT (IMPLIES (NORMALP H G) (GROUPP (QUOTIENT H G))))
```

It is easily shown that any subgroup of index 2 is normal. For example, by direct computation,

(normalp (alt 5) (sym 5)) = T.

As another example, the element (1 2 0) of (sym 3) generates a subgroup of order 3 in a group of order 6, and therefore,

(normalp (subgroup '((0 1 2) (1 2 0) (2 0 1)) (sym 3))) = T.

According to GROUPP-QUOTIENT, its quotient group

```
(quotient (sym 3) (subgroup '((0 1 2) (1 2 0) (2 0 1)) (sym 3)))
```

is a group of order 2:

```
((((0 1 2) (1 2 0) (2 0 1)) ((0 2 1) (1 0 2) (2 1 0)))
 (((0 2 1) (1 0 2) (2 1 0)) ((0 1 2) (1 2 0) (2 0 1))))
```

Of course, any subgroup of an abelian group is normal. For example, `(subgroup '(1 3 9) (z* 13))` is a normal subgroup of `(z* 13)` of index 4. Its quotient group is

```
(((1 3 9) (2 5 6) (7 8 11) (4 10 12))
 ((2 5 6) (4 10 12) (1 3 9) (7 8 11))
 ((7 8 11) (1 3 9) (4 10 12) (2 5 6))
 ((4 10 12) (7 8 11) (2 5 6) (1 3 9)))
```

6 Parametrized Subgroups

The macro `defsubgroup` calls `defgroup` to define a subgroup of a given group `g`. The last two arguments of `defgroup` are not supplied to `defsubgroup`, since they are always `(op x y g)` and `(inv x g)`. As an illustration, the *centralizer* of an element `a` of `g` is the subgroup consisting of all elements that commute with `a`. The definition of its element list, `(centizer-elts a g)`, is straightforward. Several of the rewrite rules required by `defgroup` are generated by `defsubgroup`, but the following must be proved by the user:

```
(defthm dlistp-centizer-elts
  (implies (and (groupp g) (in a g)) (dlistp (centizer-elts a g))))
(defthm sublistp-centizer-elts
  (implies (and (groupp g) (in a g)) (sublistp (centizer-elts a g) (elts g))))
(defthm centizer-elts-identity
  (implies (and (groupp g) (in a g)) (equal (car (centizer-elts a g)) (e g))))
(defthm consp-centizer-elts
  (implies (and (groupp g) (in a g)) (consp (centizer-elts a g))))
(defthm centizer-elts-closed
  (implies (and (groupp g) (in a g)
                (member-equal x (centizer-elts a g))
                (member-equal y (centizer-elts a g)))
           (member-equal (op x y g) (centizer-elts a g))))
(defthm centizer-elts-inverse
  (implies (and (groupp g) (in a g) (member-equal x (centizer-elts a g)))
           (member-equal (inv x g) (centizer-elts a g))))
```

we may then invoke

```
(defsubgroup centralizer (a g) (and (groupp g) (in a g)) (centizer-elts a g))
```

In addition to the lemmas generated by `defsubgroup`, this produces

```
(DEFTHM SUBGROUPP-CENTRALIZER
  (IMPLIES (AND (GROUPP G) (IN A G))
           (SUBGROUPP (CENTRALIZER A G) G)))
```

The *center* of `g` consists of all elements that commute with every element of `g`. The list of such elements, `(cent-elts g)` is again easily defined, and after proving the requisite rewrite rules, we have

```
(defsubgroup center (g) (groupp g) (cent-elts g))
```

Our final example is the cyclic subgroup generated by an element `a` of `g`. First we define the powers of `a`:

```
(defun power (a n g)
  (if (zp n)
      (e g)
    (op a (power a (1- n) g) g)))
```

The usual formulas for a product of powers and a power of a power are derived by induction:

```
(defthm power+
  (implies (and (groupp g) (in a g) (natp n) (natp m))
           (equal (op (power a n g) (power a m g) g)
                  (power a (+ n m) g))))
(defthm power*
  (implies (and (groupp g) (in a g) (natp n) (natp m))
           (equal (power (power a n g) m g)
                  (power a (* n m) g))))
```

Next, we define the order of a in g:

```
(defun ord-aux (a n g)
  (declare (xargs :measure (nfix (- (order g) n))))
  (if (equal (power a n g) (e g))
      n
    (if (and (natp n) (< n (order g)))
        (ord-aux a (1+ n) g)
      ())))
(defun ord (a g) (ord-aux a 1 g))
```

We cannot have `(ord a g)` = `NIL`, for it would then follow from `power+` that the powers of `a` include `(order g)` $+ 1$ distinct elements. This observation has the following consequences:

```
(defthm ord<=order
  (implies (and (groupp g) (in a g))
           (and (posp (ord a g)) (<= (ord a g) (order g)))))
(defthm divides-ord
  (implies (and (groupp g) (in a g) (natp n))
           (iff (equal (power a n g) (e g)) (divides (ord a g) n))))
(defthm power-mod
  (implies (and (groupp g) (in a g) (natp n))
           (equal (power a n g) (power a (mod n (ord a g)) g))))
(defthm ord-power-div
  (implies (and (groupp g) (in a g) (posp n) (divides n (ord a g)))
           (equal (ord (power a n g) g) (/ (ord a g) n))))
```

Thus, there are `(ord a g)` distinct powers of `a`. A list of these elements is computed by `powers`:

```
(defun powers-aux (a n g)
  (if (zp n)
      ()
    (append (powers-aux a (1- n) g) (list (power a (1- n) g)))))
(defun powers (a g) (powers-aux a (ord a g) g))
```

The following are readily derived from the definition:

```
(defthm member-powers
  (implies (and (groupp g) (in a g)
                (natp n) (< n (ord a g)))
           (equal (nth n (powers a g))
                  (power a n g))))
```

```
(defthm power-index
  (implies (and (groupp g) (in a g)
                (member-equal x (powers a g)))
           (equal (power a (index x (powers a g)))
                  x)))
```

It follows from `power-mod` that `(powers a g)` is closed under the group operation, and it follows from `power-index` and `power+` that for `x` in `(powers a g)`,

```
(inv x g) = (power a (- (ord a g) (index x (powers a g))) g)
```

and hence, `(inv x g)` belongs to `(powers a g)`. The remaining prerequisites are trivial, and we have

```
(defsubgroup cyclic (a g)
  (and (groupp g) (in a g))
  (powers a g))
```

Note that the two example subgroups at the end of Section 5 can be computed as cyclic subgroups:

`(subgroup '((0 1 2) (1 2 0) (2 0 1)) (sym 3))` = `(cyclic '(1 2 0) (sym 3))`

and

`(subgroup '(1 3 9) (z* 13))` = `(cyclic 3 (z* 13))`.

As another example, the permutation `(1 2 3 4 0)` of `(sym 5)` is of order 5, and its cyclic subgroup

`(cyclic '(1 2 3 4 0) (sym 5))`

is

```
(((0 1 2 3 4) (1 2 3 4 0) (2 3 4 0 1) (3 4 0 1 2) (4 0 1 2 3))
 ((1 2 3 4 0) (2 3 4 0 1) (3 4 0 1 2) (4 0 1 2 3) (0 1 2 3 4))
 ((2 3 4 9 1) (3 4 0 1 2) (4 0 1 2 3) (0 1 2 3 4) (1 2 3 4 0))
 ((3 4 0 1 2) (4 0 1 2 3) (0 1 2 3 4) (1 2 3 4 0) (2 3 4 0 1))
 ((4 0 1 2 3) (0 1 2 3 4) (1 2 3 4 0) (2 3 4 0 1) (3 4 0 1 2)))
```

7 Abelian Case of Cauchy's Theorem

The formulation of Cauchy's Theorem requires a witness function, which searches a group for an element of a given order:

```
(defun elt-of-ord-aux (l p n)
  (if (consp l)
      (if (= (ord (car l) g) n)
          (car l)
        (elt-of-ord-aux (cdr l) n g))
    ()))
(defun elt-of-ord (n g) (elt-of-ord-aux (elts g) n g))
```

Thus, `(elt-of-ord n g)` selects an element of g of order `n`, or returns `NIL` if none exists:

```
(defthm elt-of-ord-ord
  (implies (and (groupp g) (natp n) (elt-of-ord n g))
           (and (in (elt-of-ord n g) g)
                (equal (ord (elt-of-ord n g) g)
                       n))))
(defthm elt-of-ord-ord
  (implies (and (groupp g) (natp n) (null (elt-of-ord n g)) (in a g))
           (not (= (ord a g) n))))
```

Here are some simple examples:

- `(elt-of-ord 5 (sym 5))` = `(1 2 3 4 0)`;
- `(elt-of-ord 22 (z* 23))` = 5, the least primitive root of 23;
- `(elt-of-ord (order (z* 35)) (z* 35))` = `NIL`, since `(z* 35)` is not cyclic.

The theorem may be stated as follows:

```
(defthm cauchy
  (implies (and (groupp g)
                (primep p)
                (divides p (order g)))
           (and (in (elt-of-ord p g) g)
                (equal (ord (elt-of-ord p g) g) p))))
```

Our proof, which closely adheres to the informal proof presented in [5], consists of two steps, both involving induction on the order of `g`. First, it is proved for abelian `g`, and then it is shown that if `g` is nonabelian, then it must have a proper subgroup with order divisible by `p`. Both steps depend critically on a result from `euclid.lisp`:

```
(defthm euclid
    (implies (and (primep p)
                  (integerp a)
                  (integerp b)
                  (not (divides p a))
                  (not (divides p b)))
             (not (divides p (* a b)))))
```

If `g` has no element of order `p`, then by `ord-power-div`, it has no element of order divisible by `p`, and hence no cyclic subgroup of order divisible by `p`. Combining `lagrange` and `euclid`, we have

```
(defthm divides-order-quotient
  (implies (and (groupp g)
                (primep p)
                (divides p (order g))
                (not (elt-of-ord p g))
                (in a g))
           (divides p (order (quotient g (cyclic a g))))))
```

By `QUOTIENT-OP-REWRITE` and induction,

```
(defthm lcoset-power
  (implies (and (normalp h g) (in x g) (natp n))
           (equal (power (lcoset x h g) n (quotient g h))
                  (lcoset (power x n g) h g))))
```

It follows that

`(power (lcoset x h g) (ord x g) (quotient g h))` = `(lcoset (e g) h g)`

where

`(lcoset (e g) h g)` = `(e (quotient g h))`

By `divides-ord`, `(ord x g)` is divisible by `(ord (lcoset x h g) (quotient g h))`. Therefore, if `g` has no element of order divisible by an integer `m`, then neither does `(quotient g h)`:

```
(defthm lift-elt-of-ord
  (implies (and (normalp h g)
                (posp m)
                (elt-of-ord m (quotient g h)))
           (elt-of-ord m g)))
```

Now assume that g is abelian. Then every subgroup of g is abelian and normal. Since the quotient group of any nontrivial cyclic subgroup of g has a smaller order than g, we have our induction scheme:

```
(defun cauchy-induction (g)
  (declare (xargs :measure (order g)))
  (if (and (groupp g)
           (abelianp g)
           (> (order g) 1))
      (cauchy-induction (quotient g (cyclic (cadr (elts g)) g)))
    ()))

(defthm cauchy-abelian-lemma
  (implies (and (groupp g)
                (abelianp g)
                (primep p)
                (divides p (order g)))
           (elt-of-ord p g))
  :hints (("Goal" :induct (cauchy-induction g))))
```

In the proof of the above lemma, ACL2 generates a single nontrivial subgoal, the hypothesis of which is the instantiation of the goal with

`(quotient g (cyclic (cadr (elts g)) g))`

substituted for g. By `divides-order-quotient`, the order of this quotient group is divisible by `p`, and therefore, by hypothesis, it has an element of order `p`. The subgoal follows from `lift-elt-of-ord`.

The final theorem is an immediate consequence of `cauchy-abelian-lemma` and `elt-of-ord-ord`:

```
(defthm cauchy-abelian
  (implies (and (groupp g)
                (abelianp g)
                (primep p)
                (divides p (order g)))
           (and (in (elt-of-ord p g) g)
                (equal (ord (elt-of-ord p g) g) p))))
```

8 Conjugacy Classes and the Class Equation

The general case of Cauchy's Theorem ia based on an expression for the order of a group derived from a partition of its elements into *conjugacy classes*. The orsered list of conjugates of `x` is `(conjs x g)`, computed by

```
(defun conjs-aux (x l g)
  (if (consp l)
      (if (member-equal (conj x (car l) g)
                        (conjs-aux x (cdr l) g))
  (conjs-aux x (cdr l) g)
        (insert (conj x (car l) g)
        (conjs-aux x (cdr l) g)
               g))
    ()))
(defund conjs (x g) (conjs-aux x (elts g) g))
```

Conjugacy is easily shown to be an equivalence relation, and it follows that intersecting classes are equal:

```
(defthmd equal-conjs
  (implies (and (groupp g) (in x g) (in y g)
                (member-equal y (conjs x g)))
           (equal (conjs y g) (conjs x g))))
```

We define a bijection between the conjugates of x and the cosets of its centralizer:

```
(defund conj2coset (y x g) (lcoset (inv (conjer y x g) g) (centralizer x g) g))
(defund coset2conj (c x g) (conj x (inv (car c) g) g))
(defthm coset2conj-conj2coset
  (implies (and (groupp g) (in x g) (member-equal y (conjs x g)))
           (equal (coset2conj (conj2coset y x g) x g)
                  y)))
(defthm conj2coset-coset2conj
  (implies (and (groupp g) (in x g) (member-equal c (lcosets (centralizer x g) g)))
           (equal (conj2coset (coset2conj c x g) x g)
                  c)))
```

It follows that the size of the conjugacy class is the index of the centralizer:

```
(defthm len-conjs-cosets
(implies (and (groupp g) (in x g))
         (equal (len (conjs x g))
                (subgroup-index (centralizer x g) g))))
```

Since `(len (conjs x g))` = 1 iff `(in x (center g))`, a list of the nontrivial conjugacy classes is computed by

```
(defun conjs-list-aux (l g)
  (if (consp l)
      (let ((conjs (conjs-list-aux (cdr l) g)))
        (if (or (in (car l) (center g))
                (member-list (car l) conjs))
            conjs
          (cons (conjs (car l) g) conjs)))
    ()))
(defund conjs-list (g) (conjs-list-aux (elts g) g))
```

Thus, we can show that the following is a list of distinct elements that contains every element of g:

```
(append (elts (center g)) (append-list (conjs-list g))))
```

As a consequence, we have the *class equation*:

```
(defthmd class-equation
  (implies (groupp g)
           (equal (len (append (elts (center g)) (append-list (conjs-list g))))
                  (order g))))
```

9 General Case of Cauchy's Theorem

Assume that the order of g is divisible by a prime p. The function `find-elt` searches for an element outside the center of g that has a centralizer with order divisible by p:

```
(defun find-elt-aux (l g p)
  (if (consp l)
      (if (and (not (in (car l) (center g)))
               (divides p (order (centralizer (car l) g))))
          (car l)
        (find-elt-aux (cdr l) g p))
    ()))
(defund find-elt (g p) (find-elt-aux (elts g) g p))
```

If such an element exists, then since it is not in the center, the order of its centralizer is less than that of g. This observation provides our induction scheme:

```
(defun cauchy-induction (g p)
  (declare (xargs :measure (order g)))
  (if (and (groupp g) (primep p) (find-elt g p))
      (cauchy-induction (centralizer (find-elt g p) g) p)
    t))
```

On the other hand, if no such element exists, then for every non-central element `x`, `lagrange` implies that the index of the centralizer of `x` is divisible by `p`, and by `len-conjs-cosets`, so is `(len (conjs x g))`. According to the class equation, the same is true of `(center g)`. Since `(center g)` is abelian, we may apply `cauchy-abelian` to complete the induction, and we have

```
(defthmd cauchy
  (implies (and (groupp g) (primep p) (divides p (order g)))
           (and (in (elt-of-ord p g) g)
                (equal (ord (elt-of-ord p g) g) p))))
```

10 Conclusion

In 2007, Georges Gonthier et al. [2] embarked on a formalization of finite group theory in Coq, with the objective of a machine-checked proof of the Feit-Thompson Theorem: *All groups of odd order are solvable.* Six years later, the ultimate success of this undertaking was announced in an Inria Technical Report [1] listing fifteen coauthors. In light of the experience of our present project, it would be unsurprising to find that the Inria result did indeed involve some ninety man-years of effort.

The leap in complexity from Cauchy's Theorem to Feit-Thompson is daunting, but as C. S. Lewis reminds us, "With the possible exception of the equator, everything begins somewhere." We may find further solace in a plan to pursue a direction orthogonal to Inria's objective of the classification of finite groups, and better suited to ACL2's strengths. While not specifically designed for the formalization of higher mathematics, ACL2 is equipped with sophisticated procedures for managing rational arithmetic and polynomials. [4] Algebraic number theory, the study of finite extension fields of the rationals and their Galois groups, could be a fruitful application area founded on a formalization of elementary group theory. We have already demonstrated that our approach provides group computations in a straightforward manner, and we may anticipate that the computational power and proof automation of ACL2 can be brought to bear on the analysis and verification of a variety of number-theoretic algorithms of practical significance. Clearly, there is much work to be done before such a plan can be more than a fanciful dream.

References

[1] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi & Laurent Théry (2013): *A Machine-Checked Proof of the Odd Order Theorem*. In: *International Conference on Interactive Theorem Proving*, pp. 163–179, doi:10.1017/S0960129511000132.

[2] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi & Laurent Théry (2007): *A Modular Formalisation of Finite Group Theory*. Technical Report RR-6156, Inria.

[3] Jónathan Heras, Francisco Martín-Mateos & Vico Pascual (2015): *Modelling Algebraic Structures and Morphisms in ACL2*. *Applicable algebra in engineering, communication and computing* 26(3), pp. 277–303, doi:10.1007/s00200-015-0252-9.

[4] Warren Hunt, Robert Krug & J Moore (2003): *Linear and Nonlinear Arithmetic in ACL2*. In Daniel Geist & Enrico Tronci, editors: *Correct Hardware Design and Verification Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 319–333, doi:10.1145/2422.322411.

[5] Joseph Rotman (1965): *The Theory of Groups: An Introduction*. Allyn and Bacon.

[6] Eric Smith: *A Formalization of Groups*. `books/kestrel/algebra/groups-encap.lisp`, ACL2 repository.

[7] Yuan Yu (1990): *Computer Proofs in Group Theory*. *Journal of Automated Reasoning* 6(3), doi:10.1007/BF00244488.