A Formalization of Finite Group Theory: Part II

David M. Russinoff
david@russinoff.com

This is the second installment of an exposition of an ACL2 formalization of finite group theory. The first, which was presented at the 2022 ACL2 workshop, covered groups and subgroups, cosets, normal subgroups, and quotient groups, culminating in a proof of Cauchy's Theorem: *If the order of a group G is divisible by a prime p, then G has an element of order p.* This sequel addresses homomorphisms, direct products, and the Fundamental Theorem of Finite Abelian Groups: *Every finite abelian group is isomorphic to the direct product of a list of cyclic p-groups, the orders of which are unique up to permutation.* This theorem is a suitable application of ACL2 because of its extensive reliance on recursion and induction as well as the constructive nature of the factorization. The proof of uniqueness is especially challenging, requiring the formalization of vague intuition that is commonly taken as self-evident.

1 Introduction

In comparison to higher-order logic theorem provers, ACL2 offers a high degree of proof automation at the expense of logical expressiveness. With regard to the formalization of pure mathematics, basic concepts are often difficult to formulate in a first-order logic, but when this obstacle is overcome, proofs are relatively straightforward. This prospect has motivated our pursuit of an ACL2 formalization of abstract algebra, beginning with finite group theory, an area in which substantial progress has already been achieved with other provers, most notably the Coq proof assistant [1]. Our investigation of this subject, which deals with properties of operations defined on sets, must address two limitations of the ACL2 logic: (1) quantification over functions is not provided, and (2) ACL2 data are ordered lists rather than sets. A common solution to these problems is the use of constrained functions. In particular, a natural approach to the formalization of group theory begins with an `encapsulate` form that introduces a set of constrained functions including a predicate representing group membership, a binary group operation, and a unary inverse operator. An alternative scheme based on `defn-sk` was devised by Yuan Yu in his 1990 Nqthm formalization [4], which included a proof of Lagrange's Theorem: *The order of a group is divisible by that of any subgroup.*

Our original submission on this subject to the 2022 ACL2 Workshop [3] was motivated by the observation that any significant progress beyond Lagrange's Theorem would require the facility of proof by induction on the order of a group, which is apparently unavailable through either of the methods mentioned above. That is, a more productive ACL2 formalization of group theory would begin with an explicit predicate that recognizes groups of arbitrary well-defined orders. (This approach necessarily limits the investigation to finite groups.) Thus, the predicate `groupp` defines a group of order n to be an $n \times n$ matrix (a list of n rows of length n) representing the group's operation table, which is stipulated to satisfy the usual group axioms. The first row of the matrix is the list of group elements, the order of which is insignificant except that the first element must be the identity:

```
(defmacro elts (g) `(car ,g))
(defmacro in (x g) `(member-equal ,x (elts ,g)))
(defund e (g) (caar g))
```

Submitted to:
ACL2 Workshop 2023

The index of an element x of g, (ind x g), is its position in the element list:

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

Thus, the group operation is defined by

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

The lack of ACL2 support for unordered sets is mitigated by exploiting the notion of an ordered list of elements of a group. Thus, for example, a left coset of a subgroup is defined to be ordered with respect to the larger group, thereby ensuring that intersecting cosets are equal. Note that a subgroup need not be ordered with respect to its parent. For example, a cyclic subgroup has a natural ordering that is generally different from that of the parent group. In most cases, however, we arrange for the ordering to be inherited.

In order to circumvent the cumbersome explicit construction of the defining table for every group of interest, we introduce a `defgroup` macro that generates a parametrized group definition and proofs of the axioms (through functional instantiation) once the user has supplied the element list and terms specifying the binary operation and the inverse operator. This is used, for example, in the definition of the quotient group of a normal subgroup as well as the symmetric groups. A similar `defsubgroup` macro facilitates the definition of parametrized subgroups, e.g., the centralizer of a group element, the center of a group, cyclic subgroups, and intersections of subgroups. As a proof of concept, the culmination of the 2022 submission is 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.* The theory up to this point is embodied in the first four books of the ACL2 directory `books/projects/groups`: `lists`, `groups`, `quotients`, and `cauchy`.

The 2022 paper is a prerequisite for a reading of this sequel, which presents a continuation of the theory comprising three additional books of the `groups` directory. The first of these, `maps`, addresses another class of functions that must be encoded in the logic: group homomorphisms (Section 2), which we represent as association lists. The second, `products`, covers direct products (Section 3). The results of these two books are applied in the third, `abelian`, in a proof of the Fundamental Theorem of Finite Abelian Groups: *Every finite abelian group is isomorphic to the direct product of a list of cyclic p-groups, the orders of which are unique up to permutation.* The proof also applies various number-theoretic results from the book `projects/numbers/euclid`. The proof is in three parts: (1) the factorization of an abelian p-group as a product of cyclic groups (Section 4, (2) the factorization of an arbitrary abelian group as a product of p-groups (Section 5), and (3) the uniqueness of the factorization (Section 6). This theorem is a suitable application of ACL2 because of its extensive reliance on recursion and induction as well as the constructive nature of the factorization. The proof of uniqueness is especially challenging, requiring the formalization of vague intuition that is usually taken as self-evident [2].

2 Homomorphisms

A *homomorphism* is a function `f` from a group `g` to a group `h` that preserves the group operation, i.e., satisfies the identity `(op x y g) = (op (f x) (f y) h)`, and thus relates the algebraic properties of `g` and `h`. Of particular interest is the case of a bijective homomorphism, or an *isomorphism*, which establishes the algebraic equivalence of two groups.

In order to introduce group homomorphisms into our theory, we define a *map* to be an alist with pairwise distinct cars:

```
(defun cons-listp (m)
  (if (consp m) (and (consp (car m)) (cons-listp (cdr m))) (null m)))
(defund domain (m) (strip-cars m))
(defund mapp (m) (and (cons-listp m) (dlistp (domain m))))
```

The function `mapply` applies a map to an element of its domain:

```
(defund mapply (map x) (cdr (assoc-equal x map)))
```

The macro `defmap` provides a convenient method of defining maps. The macro call

(`defmap` *name args domain val*)

defines a family of maps parametrized by *args* with given *domain*, which is assumed to be a dlist. The parameter *val* is the value that the map assigns to an element x of its domain. For example, the following form automates the construction of a composition of two maps:

```
(defmap compose-maps (map2 map1)
  (domain map1)
  (mapply map2 (mapply map1 x)))
```

Evaluation of this form generates two definitions and proves three lemmas:

```
(DEFUN COMPOSE-MAPS-AUX (L MAP2 MAP1)
  (IF (CONSP L)
      (LET ((X (CAR L)))
        (CONS (CONS X (MAPPLY MAP2 (MAPPLY MAP1 X)))
              (COMPOSE-MAPS-AUX (CDR L) MAP2 MAP1)))
    NIL))
(DEFUN COMPOSE-MAPS (MAP2 MAP1)
  (COMPOSE-MAPS-AUX (DOMAIN MAP1) MAP2 MAP1))
(DEFTHM DOMAIN-COMPOSE-MAPS
  (IMPLIES (DLISTP (DOMAIN MAP1))
           (EQUAL (DOMAIN (COMPOSE-MAPS MAP2 MAP1))
                  (DOMAIN MAP1))))
(DEFTHM MAPP-COMPOSE-MAPS
  (IMPLIES (DLISTP (DOMAIN MAP1))
           (MAPP (COMPOSE-MAPS MAP2 MAP1))))
(DEFTHM COMPOSE-MAPS-VAL
  (IMPLIES (MEMBER-EQUAL X (DOMAIN MAP1))
           (EQUAL (MAPPLY (COMPOSE-MAPS MAP2 MAP1) X)
                  (MAPPLY MAP2 (MAPPLY MAP1 X)))))
```

A *homomorphism* from a group `g` to a group `h` is a map that satisfies the following predicate:

```
(defund homomorphismp (map g h)
  (and (groupp g)
       (groupp h)
       (mapp map)
       (sublistp (elts g) (domain map))
       (equal (mapply map (e g)) (e h))
       (not (codomain-cex map g h))
       (not (homomorphism-cex map g h))))
```

The functions codomain-cex and homomorphism-cex search for counterexamples of these two properties:

```
(implies (in x g)
         (in (mapply map x) h))
(implies (and (in x g) (in y g))
         (equal (mapply map (op x y g))
                (op (mapply map x) (mapply map y) h)))
```

Once these two implications have been proved to hold for a proposed homomorphism, the corresponding conjuncts of the definition are readily established. In this manner, it is easily shown, for example, that a composition of homomorphisms is a homomorphism:

```
(defthm homomorphismp-compose-maps
  (implies (and (homomorphismp map1 g h) (homomorphismp map2 h k))
           (homomorphismp (compose-maps map2 map1) g k)))
```

The *image* of a homomorphism map from g to h is a subgroup of h. Its element list is defined using the function insert, which ensures that it is ordered with respect to h:

```
(defun ielts-aux (map l h)
  (if (consp l)
      (insert (mapply map (car l))
              (ielts-aux map (cdr l) h)
              h)
    ()))
(defund ielts (map g h)
  (ielts-aux map (elts g) h))
(defthm ordp-ielts
  (implies (homomorphismp map g h)
           (ordp (ielts map g h) h)))
```

The group (image g h) is automatically defined by defsubgroup once the usual prerequisite lemmas have been proved:

```
(defthm dlistp-ielts
  (implies (homomorphismp map g h)
           (dlistp (ielts map g h))))
(defthm sublistp-ielts
  (implies (homomorphismp map g h)
           (sublistp (ielts map g h) (elts h))))
(defthm consp-ielts
  (implies (groupp g)
           (consp (ielts map g h))))
(defthm ielts-closed
  (implies (and (homomorphismp map g h)
                (member-equal x (ielts map g h))
                (member-equal y (ielts map g h)))
           (member-equal (op x y h) (ielts map g h))))
(defthm ielts-inverse
  (implies (and (homomorphismp map g h)
                (member-equal x (ielts map g h)))
           (member-equal (inv x h) (ielts map g h))))
```

The arguments of defsubgroup are the subgroup parameters, the parent group, the constraint that must be satisfied by the parameters, and the element list:

```
(defsubgroup image (map g) h
  (homomorphismp map g h)
  (ielts map g h))
```

The *kernel* of `map` is a subgroup of `g` consisting of those elements that are mapped to the identity element of `h`:

```
(defun kelts-aux (map l h)
  (if (consp l)
      (if (equal (mapply map (car l)) (e h))
          (cons (car l) (kelts-aux map (cdr l) h))
        (kelts-aux map (cdr l) h))
  ()))
(defund kelts (map g h)
  (kelts-aux map (elts g) h))
```

Once again, we invoke `defsubgroup` after establishing its prerequisite lemmas:

```
(defsubgroup kernel (map h) g
  (homomorphismp map g h)
  (kelts map g h))
```

The usual classes of homomorphisms are defined in terms of the image and the kernel:

```
(defund epimorphismp (map g h)
  (and (homomorphismp map g h)
       (equal (image map g h) h)))
(defund endomorphismp (map g h)
  (and (homomorphismp map g h)
       (equal (kernel map h g) (trivial-subgroup g))))
(defund isomorphismp (map g h)
  (and (epimorphismp map g h) (endomorphismp map g h)))
```

The inverse of an isomorphism is defined by `defmap`:

```
(defun preimage-aux (x map l)
  (if (consp l)
      (if (equal x (mapply map (car l)))
  (car l)
 (preimage-aux x map (cdr l)))
    ()))
(defund preimage (x map g)
  (preimage-aux x map (elts g)))
(defmap inv-isomorphism (map g h) (elts h) (preimage x map g))
(defthmd isomorphismp-inv
  (implies (isomorphismp map g h)
           (isomorphismp (inv-isomorphism map g h) h g)))
```

We shall also require the following important property of isomorphisms:

```
(defthm isomorphismp-compose-maps
  (implies (and (isomorphismp map1 g h) (isomorphismp map2 h k))
                (isomorphismp (compose-maps map2 map1) g k)))
```

3 Direct Products

Direct products provide a means of constructing complex groups from simpler ones. Given a non-null proper list of groups `l`, we shall define the group `(direct-product l)`. Its element list is the Cartesian product `(group-tuples l)`, defined as follows:

```
(defun conses (x l)
  (if (consp l)
      (cons (cons x (car l)) (conses x (cdr l)))
    ()))
(defun group-tuples-aux (l m)
  (if (consp l)
      (append (conses (car l) m)
              (group-tuples-aux (cdr l) m))
    ()))
(defun group-tuples (l)
  (if (consp l)
      (group-tuples-aux (elts (car l)) (group-tuples (cdr l)))
    (list ())))
```

It is easily shown that the length of `(group-tuples l)` is the product of the orders of the members of `l`. If `x` is a member of this list, then `x` is a list of the same length as `l`, and each member of `x` is a group element of the corresponding member of `l`. The `car` of `(group-tuples l)`, which will be the identity element of the direct product, is the list defined as follows:

```
(defun e-list (l)
  (if (consp l)
      (cons (e (car l)) (e-list (cdr l)))
    ()))
```

The group operation and inverse operator are defined recursively:

```
(defun dp-op (x y l)
  (if (consp l)
      (cons (op (car x) (car y) (car l))
            (dp-op (cdr x) (cdr y) (cdr l)))
    ()))
(defun dp-inv (x l)
  (if (consp l)
      (cons (inv (car x) (car l))
            (dp-inv (cdr x) (cdr l)))
    ()))
```

Once the requisite group properties are proven, we invoke `defgroup` to construct the direct product:

```
(defgroup direct-product (l)
  (and (group-list-p l) (consp l)) ;parameter constraint
  (group-tuples l)                 ;element list
  (dp-op x y l)                    ;group operation
  (dp-inv x gl))                   ;inverse operator
```

The ordering of the elements of the direct product is given by the following, which is useful in proving that a given subgroup is ordered:

```
(defthmd ind-dp-compare
  (implies (and (group-list-p l)
                (consp l)
                (in x (direct-product l))
                (in y (direct-product l)))
           (iff (< (ind x (direct-product l))
                   (ind y (direct-product l)))
                (or (< (ind (car x) (car l))
                       (ind (car y) (car l)))
```

```
                    (and (consp (cdr l))
                         (equal (car x) (car y))
                         (< (ind (cdr x) (direct-product (cdr l)))
                            (ind (cdr y) (direct-product (cdr l)))))))))
```

A construction related to the direct product is the list of products of elements of two subgroups h and k of a group g. Note that the definition ensures that this list is ordered with respect to g:

```
(defun products-aux (l g)
  (if (consp l)
      (insert (op (caar l) (cadar l) g) (products-aux (cdr l) g) g)
    ()))
(defund products (h k g)
  (products-aux (group-tuples (list h k)) g))
```

While (products h k g) does not in general form a subgroup of g, it does when either h or k is normal:

```
(defsubgroup product-group (h k) g
  (and (subgroupp h g)
       (subgroupp k g)
       (or (normalp h g) (normalp k g)))
  (products h k g))
```

When h and k are both normal in g, so is (product-group (h k g).

We have the following formula for the length of (products h k g):

```
(defthmd len-products
  (implies (and (subgroupp h g)
                (subgroupp k g))
           (equal (len (products h k g))
                  (/ (* (order h) (order k))
                     (order (group-intersection h k g))))))
```

The derivation of this formula is based on the following function, which converts a list l of members of (lcosets (group-intersection h k g) h) to a list of members of (lcosets k g) by replacing each member c of l with (lcoset (car c) k g):

```
(defun lift-cosets-aux (l k g)
  (if (consp l)
      (cons (lcoset (caar l) k g)
            (lift-cosets-aux (cdr l) k g))
    ()))
```

We apply lift-cosets-aux to the full list (lcosets (group-intersection h k g) g):

```
(defund lift-cosets (h k g)
  (lift-cosets-aux (lcosets (group-intersection h k g) h) k g))
```

The result is a list of distinct elements of (lcosets k g), and therefore, appending them yields a dlist:

```
(defthm dlistp-append-list-lift-cosets
  (implies (and (subgroupp h g) (subgroupp k g))
           (dlistp (append-list (lift-cosets h k g)))))
```

The length of (lift-cosets h k g) is that of (lcosets (group-intersection h k g) h), which is the quotient

```
                  (/ (order h) (order (intersect-groups h k g)))
```

Since the length of each member of the list is `(order k)`, we have the following expression for the length of the appended list:

```
(defthm len-append-list-lift-cosets
  (implies (and (subgroupp h g) (subgroupp k g))
           (equal (len (append-list (lift-cosets h k g)))
                  (/ (* (order h) (order k))
                     (order (group-intersection h k g))))))
```

It is easy to show that each member of `(lift-cosets h k g)` is a sublist of `(products h k g)`. On the other hand, if `x` is an element of `(products h k g)`, then `x = (op a b g)`, where `a` is in `h` and `b` is in `k`. Some member of (lcosets (group-intersection h k g) h) contains `a`, as does the corresponding member of `(lift-cosets h k g)`, which therefore contains `x`. Thus, `(products h k g)` is a sublist of `(append-list (lift-cosets h k g))`. Since both lists are dlists and each is a sublist of the other, they have the same length, and the formula `len-products` follows from `len-append-list-lift-cosets`.

The product of a list of subgroups is defined recursively:

```
(defun product-group-list (l g)
  (if (consp l)
      (product-group (car l) (product-group-list (cdr l) g) g)
    (trivial-subgroup g)))
```

By induction, if each group in `l` is normal in g, then so is `(product-group-list l g)`. If `l` satisfies the following predicate, then that subgroup is isomorphic to `(direct-product l)`:

```
(defun internal-direct-product-p (l g)
  (if (consp l)
      (and (internal-direct-product-p (cdr l) g)
           (normalp (car l) g)
           (equal (group-intersection (car l) (product-group-list (cdr l) g) g)
                  (trivial-subgroup g)))
    (null l)))
```

Moreover, if that subgroup has the same order as g, then since it inherits the ordering of g, the two groups are equal. The isomorphism is conveniently constructed by `defmap`:

```
(defun product-list-val (x g)
  (if (consp x)
      (if (consp (cdr x))
          (op (car x) (product-list-val (cdr x) g) g)
        (car x))
    ()))
(defmap product-list-map (l g)
  (group-tuples l)
  (product-list-val x g))
(defthmd isomorphismp-dp-idp
  (implies (and (groupp g)
                (consp l)
                (internal-direct-product-p l g)
                (= (product-orders l) (order g)))
           (isomorphismp (product-list-map l g)
                         (direct-product l)
                         g)))
```

Note also that the product of two non-intersecting internal direct products is an internal direct product:

```
(defthmd internal-direct-product-append
  (implies (and (internal-direct-product-p l g)
                (internal-direct-product-p m g)
                (equal (group-intersection (product-group-list l g)
                                           (product-group-list m g)
                                           g)
                       (trivial-subgroup g)))
           (internal-direct-product-p (append l m) g)))
```

4 Factorization of p-Groups

A group is *abelian* if its operation is commutative. The notion of an abelian group may be viewed as an abstraction of the familiar numerical groups: the additive groups of integers, modular integers, rationals, and reals, and the multiplicative groups of the non-zero rationals and reals. Finite abelian groups admit a particularly simple classification as direct products of cyclic groups. We begin with the case of an abelian `p-group`, the order of which is a power of a prime *p*:

```
(defund p-groupp (g p)
  (and (primep p) (groupp g) (powerp (order g) p)))
```

In this section, we shall prove that every abelian *p*-group is an internal direct product of cyclic subgroups. This will follow by induction once we show that if such a group is not cyclic, then it is an internal direct product of two non-trivial subgroups. The proof of this result is also inductive, based on the notion of "lifting" a subgroup of a quotient group. If `n` is a normal subgroup of `g` and `h` is a subgroup of `(quotient g n)`, then `(lift h n g)` is the subgroup of `g` formed by appending the cosets that belong to `h`. In the book `groups/quotients`, we prove the following two lemmas:

```
(defthmd lift-subgroup
  (implies (and (normalp n g) (subgroupp h (quotient g n)))
           (subgroupp n (lift h n g))))

(defthmd lift-order
  (implies (and (normalp n g) (subgroupp h (quotient g n)))
           (equal (order (lift h n g)) (* (order h) (order n)))))
```

Assume `(p-groupp g p)` and `(abelianp g)`. Our objective is to construct subgroups `g1` and `g2` of `g` that satisfy the following predicate:

```
(defund desired-properties (g g1 g2)
  (and (subgroupp g1 g)
       (cyclicp g1)
       (subgroupp g2 g)
       (equal (* (order g1) (order g2)) (order g))
       (equal (group-intersection g1 g2 g) (trivial-subgroup g))))
```

The construction begins with the selection of an element `a` of maximal ord in `g`, as computed by the function `max-ord`:

```
(defun max-ord-aux (g n)
  (if (zp n)
      1
    (if (elt-of-ord n g)
        n
      (max-ord-aux g (1- n)))))
(defund max-ord (g)
  (max-ord-aux g (order g)))
```

For convenience, we collect these hypotheses in another predicate:

```
(defund phyp (a p g)
  (and (p-groupp g p)
       (abelianp g)
       (not (cyclicp g))
       (in a g)
       (equal (ord a g) (max-ord g))))
```

The first of the two subgroups is the cyclic group generated by `a`:

```
(defund g1 (a g) (cyclic a g))
```

(We shall abbreviate the term `(g1 a g)` as `g1`; related terms defined below will be similarly abbreviated.) By `cauchy`, some coset in `(quotient g g1)` has order `p`. We define `x$` to be a member of that coset:

```
(defund x$ (a p g) (car (elt-of-ord p (quotient g (g1 a g)))))
```

Thus, `x$` is not in `g1` but `(power x$ p g)` is in `g1`. This implies `(power x$ p g)` is a member of `(powers a g)`, and hence `(power x$ p g)` = `(power a i$ g)`, where `i$` is defined by

```
(defund i$ (a p g) (index (power (x$ a p g) p g) (powers a g)))
```

It follows that `i$` is divisible by `p`, for otherwise `(power a i$ g)` has the same order as `a`, implying that `x$` has order greater than `(max-ord g)`. Consider the cyclic subgroup `c$` of `g` defined as follows:

```
(defund j$ (a p g) (/ (i$ a p g) p))

(defund y$ (a p g)
  (op (power (inv a g) (j$ a p g) g)
      (x$ a p g)
      g))

(defun c$ (a p g) (cyclic (y$ a p g) g))
```

Note that `y$` is not in `g1`, but

```
(power y$ p g) = (op (power (power (inv a g) j$ g) p g) (power x$ p g) g)
               = (op (power (inv a g) (* j$ p) g) (power x$ p g) g)
               = (op (power (inv a g) i$ g) (power x$ p g) g)
               = (op (inv (power a i$ g) g) (power x$ p g) g)
               = (op (inv (power x$ p g) g) (power x$ p g) g)
               = (e g)
```

Thus, `(order c$) = (ord y$ g) = p` and `g1` and `c$` intersect trivially. Let `g*` and `a*` be defined as follows:

```
(defun g* (a p g)
  (quotient g (c$ a p g)))
(defun a* (a p g)
  (lcoset a (c$ a p g) g))
```

Then `(ord a* g*) = (max-ord g)`, for otherwise a power of a less than `(max-ord g)` would be in `c$` and therefore equal to `(e g)`. Thus, `a*` has maximal ord in `g$`. If `g*` is cyclic, then its order is `(max-ord g)`, which implies

```
(order g) = (* (order g*) p} = (* (order g1) (order c$))
```

and we may define g2 = `c$`. Otherwise we proceed by induction on `(order g`, substituting `g*` and `a*` for g and a. Let `g1*` = `(cyclic a* g*)`. By inductive hypothesis, `g*` is the internal direct product of `g1*` and some `g2*`. We define g2 to be `(lift g2* c$ g)`. This yields the following recursive definition:

```
(defund g2 (a p g)
  (declare (xargs :measure (order g)))
  (if (phyp a p g)
      (if (cyclicp (g* a p g))
          (c$ a p g)
        (lift (g2 (a* a p g) p (g* a p g))
              (c$ a p g)
              g))
    ()))
```

We need only show that `(desired-properties g g1 g2)` follows from `(desired-properties g* g1* g2*)`. We may assume that `g1` is not cyclic. We have `(order g1)` = `(max-ord g)` = `(order g1*)`, and by `lift-order`, `(order g2)` = `(* p (order g2*)`. Thus,

```
(* (order g1) (order g2)) = (* p (order g1*) (order g2*))
                          = (* p (order g*))
                          = (order g).
```

Finally, suppose `r` is in both `g1` and `g2`. Then `(lcoset r c$ g)` is in both `g1*` and `g2*`, which implies `r` is in `c$`. But then `r` is in both `g1` and `c$`, which implies `r` = `(e g)`. This completes the induction, and we have

```
(defthmd factor-p-group
  (implies (phyp a p g)
           (desired-properties g (g1 a g) (g2 a p g))))
```

This result provides the basis of the factorization of `g`. The goal is to show that `g` is the internal direct product of a list of subgroups characterized as follows:

```
(defund cyclic-p-group-p (g)
  (and (cyclicp g)
       (> (order g) 1)
       (p-groupp g (least-prime-divisor (order g)))))
(defun cyclic-p-group-list-p (l)
  (if (consp l)
      (and (cyclic-p-group-p (car l)) (cyclic-p-group-list-p (cdr l)))
    (null l)))
```

The list is constructed recursively:

```
(defun cyclic-p-subgroup-list (p g)
  (declare (xargs :measure (order g)))
  (if (and (p-groupp g p) (abelianp g) (> (order g) 1))
      (if (cyclicp g)
          (list g)
        (let ((a (elt-of-ord (max-ord g) g)))
          (cons (g1 a g) (cyclic-p-subgroup-list p (g2 a p g)))))
    ()))
```

The desired result follows from `factor-p-group` by induction on `(order g)`:

```
(defthmd p-group-factorization
  (implies (and (p-groupp g p) (abelianp g) (> (order g) 1))
           (let ((l (cyclic-p-subgroup-list p g)))
             (and (consp l)
                  (cyclic-p-group-list-p l)
                  (internal-direct-product-p l g)
                  (equal (order g) (product-orders l))))))
```

5 Factorization of Abelian Groups

We shall prove constructively that every finite abelian group is isomorphic to a direct product of cyclic p-groups. The proof is again inductive, based on `p-group-factorization` and the following lemma: *If* `(order g)` *is the product of relatively prime integers* `m` *and* `n`*, then* `g` *is the internal direct product of two subgroups of orders* `m` *and* `n`. To derive this result, we define the ordered list of all elements of g with order dividing `m`:

```
(defun elts-of-ord-dividing-aux (m l g)
  (if (consp l)
      (if (divides (ord (car l) g) m)
          (cons (car l) (elts-of-ord-dividing-aux m (cdr l) g))
        (elts-of-ord-dividing-aux m (cdr l) g))
    ()))
(defund elts-of-ord-dividing (m g)
  (elts-of-ord-dividing-aux m (elts g) g))
```

If g is abelian, then this list forms a subgroup of g:

```
(defsubgroup subgroup-ord-dividing (m) g
  (and (abelianp g) (posp m))
  (elts-of-ord-dividing m g))
```

Let `h = (subgroup-ord-dividing m g)` and `k = (subgroup-ord-dividing n g)`. If `x` is in both `h` and `k`, then `(ord x)` divides both `m` and `n`, and by `divides-gcd` of the book `euclid`, `(ord x)` divides `(gcd m n) = 1`, which implies `x = (e g)`. Thus, `h` and `k` intersect trivially:

```
(defthmd rel-prime-factors-intersection
  (implies (and (groupp g) (abelianp g)
                (posp m) (posp n) (= (gcd m n) 1))
           (let ((h (subgroup-ord-dividing m g)) (k (subgroup-ord-dividing n g)))
             (equal (group-intersection h k g) (trivial-subgroup g)))))
```

By `gcd-linear-combination` (book `euclid`), since `(gcd m n)` = 1, there exist `r` and `s` such that `(+ (* r n) (* s m))` = 1. Let x be in g. Then

```
x = (power x (+ (* r n) (* s m)) g) = (op (power x (* r n) g) (power x (* s m) g) g).
```

Since

```
(power (power x (* r n) g) m g) = (power (power x (* m n) g) r g) = (e g),
```

`(power x (* r n) g)` is in `m`, and similarly, `(power x (* s m) g)` is in `k`. Thus, by len-products,

```
(* m n) = (order g) = (len (products h k g)) = (* (order h) (order k)).
```

If p is a prime dividing `(order h)`, then by `cauchy`, `h` has an element of order p, and therefore p divides `m`, which implies p does not divide `n`. By `lagrange` and `divides-product-divides-factor` (of the book `euclid`), `(order h)` divides `m`, and therefore `(<= (order h) m)`. Similarly, `(<= (order k) n)`. Since `(* m n) = (* (order h) (order k))`, both equalities must hold:

```
(defthmd rel-prime-factors-orders
  (implies (and (groupp g) (abelianp g)
                (posp m) (posp n) (= (gcd m n) 1)
                (= (order g) (* m n)))
           (let ((h (subgroup-ord-dividing m g)) (k (subgroup-ord-dividing n g)))
             (and (equal (order h) m) (equal (order k) n)))))
```

Let `p` be the least prime divisor of `(order g)`. Let `m` be the maximum power of `p` that divides `(order g)` and let `n = (/ (order g) m)`. Then `m` and `n` are relatively prime. We define a list of subgroups of g recursively, using `cyclic-p-subgroup-list`:

```
(defun cyclic-subgroup-list (g)
  (declare (xargs :measure (order g) ))
  (if (and (groupp g) (abelianp g))
      (if (= (order g) 1)
          ()
        (let* ((p (least-prime-divisor (order g)))
               (m (max-power-dividing p (order g)))
               (n (/ (order g) m))
               (h (subgroup-ord-dividing m g))
               (k (subgroup-ord-dividing n g)))
          (append (cyclic-p-subgroup-list p h)
                  (cyclic-subgroup-list k))))
    ()))
```

The following is proved by induction, combining `rel-prime-factors-intersection` and `rel-prime-factors-orders` with `internal-direct-product-append` (Section 3) and `p-group-factorization` (Section 4):

```
(defthmd idp-cyclic-subgroup-list
  (implies (and (groupp g) (abelianp g) (> (order g) 1))
           (let ((l (cyclic-subgroup-list g)))
             (and (cyclic-p-group-list-p l)
                  (internal-direct-product-p l g)
                  (equal (product-orders l) (order g))))))
```

Finally, we invoke isomorphismp-dp-idp (Section 3):

```
(defthmd abelian-factorization
  (implies (and (groupp g) (abelianp g) (> (order g) 1))
           (let ((l (cyclic-subgroup-list g)))
             (and (cyclic-p-group-list-p l)
                  (isomorphismp (product-list-map l g) (direct-product l) g)))))
```

6 Uniqueness of the Factorization

We define the list of orders of a list of groups:

```
(defun orders (l)
  (if (consp l)
      (cons (order (car l)) (orders (cdr l)))
    ()))
```

Our objective is to show that if the direct products of two lists of cyclic p-groups `l` and `m` are isomorphic, then `(orders l)` and `(orders m)` satisfy the following predicate, which is defined in the book `lists`:

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

We shall make use of an equuivalent formulation of `permutationp`, based on a function that counts the number of occurrences of an object in a list:

```
(defun hits (x l)
  (if (consp l)
      (if (equal x (car l))
          (1+ (hits x (cdr l)))
        (hits x (cdr l)))
    0))
```

It is evident that (permutationp l m) holds iff (hits x l) = (hits x m) for all x. The formalization of this claim requires a function that searches for a counterexample:

```
(defun hits-diff-aux (test l m)
  (if (consp test)
      (if (equal (hits (car test) l) (hits (car test) m))
          (hits-diff-aux (cdr test) l m)
        (list (car test)))
    ()))
(defund hits-diff (l m) (hits-diff-aux (append l m) l m))
(defthmd hits-diff-perm (iff (permutationp l m) (not (hits-diff l m))))
```

The uniqueness proof is also based on the notion of a power of an abelian group. We define the list of nth powers of the elements of g:

```
(defun power-list-aux (l n g)
  (if (consp l)
      (insert (power (car l) n g)
              (power-list-aux (cdr l) n g)
              g)
    ()))
(defun power-list (n g)
  (power-list-aux (elts g) n g))
```

If g is abelian, then this list forms a subgroup of g:

```
(defsubgroup group-power (n) g
  (and (posp n) (groupp g) (abelianp g))
  (power-list n g))
```

If two abelian groups are isomorphic, then so are their nth powers:

```
(defthmd isomorphismp-power
  (implies (and (isomorphismp map g h) (abelianp g) (posp n))
           (isomorphismp map (group-power n g) (group-power n h))))
```

The nth power of a direct product of abelian groups is the direct product of the nth powers. The proof is more challenging than expected, as it requires showing not only that each element list is a sublist of the other, but also that both lists are ordered with respect to (direct-product l), according to the lemma ind-dp-compare (Section 3):

```
(defun group-power-list (n l)
  (if (consp l)
      (cons (group-power n (car l))
            (group-power-list n (cdr l)))
    ()))
(defthmd group-power-dp
  (implies (and (posp n) (consp l) (abelian-list-p l))
           (equal (group-power n (direct-product l))
                  (direct-product (group-power-list n l)))))
```

The nth power of a cyclic group is cyclic. For prime p, the order of (group-power p g) depends on whether p divides the order of g:

```
(defun reduce-order (n p)
  (if (divides p n) (/ n p) n))
(defthmd prime-power-cyclic
  (implies (and (cyclicp g) (primep p))
           (and (cyclicp (group-power p g))
                (equal (order (group-power p g))
                       (reduce-order (order g) p)))))
```

The list of orders of `(group-power-list p l)`:

```
(defun reduce-orders (orders p)
  (if (consp orders)
      (cons (reduce-order (car orders) p) (reduce-orders (cdr orders) p))
    ()))
(defthm order-group-power-list
  (implies (and (primep p) (cyclic-p-group-list-p l))
           (equal (orders (group-power-list p l))
                  (reduce-orders (orders l) p))))
```

It is a simple matter to identify a prime that divides at least one of the orders:

```
(defund first-prime (l) (least-prime-divisor (order (car l))))
```

Let `p = (first-prime l)`. We would like to use an induction scheme that replaces `l` and `m` with `(group-powers p l)` and `(group-powers p m)`, but in order to ensure that these lists inherit the properties of `l` and `m`, we must delete any occurrences of trivial groups:

```
(defun delete-trivial (l)
  (if (consp l)
      (if (= (order (car l)) 1)
          (delete-trivial (cdr l))
        (cons (car l) (delete-trivial (cdr l))))
    ()))
(defund reduce-cyclic (l p) (delete-trivial (group-power-list p l)))
```

Let `l' = (reduce-cyclic l p)` and `m' = (reduce-cyclic m p)`. The properties of l and m are inherited by `l'` and `m'`:

```
(defthmd reduce-cyclic-p-group-list
  (implies (and (primep p) (cyclic-p-group-list-p l))
           (cyclic-p-group-list-p (reduce-cyclic l p))))
```

We would like to show that if `(orders l')` is a permutation of `(orders m')`, then the same is true of `l` and `m`. By `hits-diff-perm`, it suffices to show that for all `x`, `(hits x (orders l)) = (hits x (orders m))`. It may be proved as a consequence of `order-group-power-list` that this holds for all `x` other than p. But note that `(orders l)` and `(orders m)` have the same product:

```
(product-orders l) = (order (direct-product l))
                   = (order (direct-product l))
                   = (product-orders m).
```

It follows that the equation holds for `x = p` as well. Thus, we have

```
(defthmd permutationp-orders
  (implies (and (consp l)
                (consp m)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (primep p)
                (isomorphismp map (direct-product l) (direct-product m))
                (permutationp (orders (reduce-cyclic l p))
                              (orders (reduce-cyclic m p))))
           (permutationp (orders l) (orders m))))
```

The base case of the induction is `(or (null l') (null m'))`. If `l' = nil`, then every element of `l` must be a group of order `p`, which then every non-trivial element of `(direct-product l)` has order `p`. But then the same must be true of `(direct-product m)` and consequently m' = nil. Therefore, if either l' or m' is null, then

```
(orders (reduce-cyclic l p)) = (orders (reduce-cyclic l p)) = nil.
```

In particular, `(permutationp (orders l') (orders m'))` and we have the following consequence of `permutationp-orders`:

```
(defthmd null-reduce-cyclic-case
  (implies (and (consp l)
                (consp m)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (primep p)
                (or (null (reduce-cyclic l p))
                    (null (reduce-cyclic m p)))
                (isomorphismp map (direct-product l) (direct-product m)))
           (permutationp (orders l) (orders m))))
```

In the remaining inductive case, we need only show that if `(direct-product l)` and `(direct-product m)` are isomorphic, then so are `(direct-product l')` and `(direct-product m')`. We begin by constructing an isomorphism between `(direct-product (group-power-list p l))` and `(direct-product l')`:

```
(defun delete-trivial-elt (x l)
  (if (consp x)
      (if (= (order (car l)) 1)
          (delete-trivial-elt (cdr x) (cdr l))
        (cons (car x) (delete-trivial-elt (cdr x) (cdr l))))
    ()))
(defmap delete-trivial-iso (l)
  (group-tuples l)
  (delete-trivial-elt x l))
(defthmd isomorphismp-delete-trivial
  (implies (and (group-list-p l) (consp (delete-trivial l)))
           (isomorphismp (delete-trivial-iso l)
                         (direct-product l)
                         (direct-product (delete-trivial l)))))
```

Now suppose `map` is an isomorphism from `(direct-product l)` to `(direct-product m)`. By `isomorphismp-power`,

```
 (isomorphismp map (group-power p (direct-product l))
                   (group-power p (direct-product m))),
```

and by `group-power-dp`,

```
 (isomorphismp map (direct-product (group-power-list p l))
                   (direct-product (group-power-list p m))).
```

Thus, the desired isomorphism is constructed as a composition of three isomorphisms:

```
(defund reduce-cyclic-iso (map l m p)
  (compose-maps
    (delete-trivial-iso (group-power-list p m))
    (compose-maps
      map
      (inv-isomorphism (delete-trivial-iso (group-power-list p l))
                       (direct-product (group-power-list p l))
                       (direct-product (reduce-cyclic l p))))))
```

We apply `isomorphismp-delete-trivial`, `isomorphismp-inv`, and `isomorphismp-compose-maps` to conclude that `reduce-cyclic-iso` is an isomorphism:

```
(defthmd isomorphismp-reduce-cyclic
  (implies (and (consp l)
                (consp m)
                (primep p)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (consp (reduce-cyclic l p))
                (consp (reduce-cyclic m p))
                (isomorphismp map (direct-product l) (direct-product m)))
           (isomorphismp (reduce-cyclic-iso map l m p)
                         (direct-product (reduce-cyclic l p))
                         (direct-product (reduce-cyclic m p)))))
```

Our theorem follows from `null-reduce-cyclic-case`, `permutationp-orders`, and `isomorphismp-reduce-cyclic` by induction:

```
(defthmd abelian-factorization-unique
  (implies (and (consp l)
                (consp m)
                (cyclic-p-group-list-p l)
                (cyclic-p-group-list-p m)
                (isomorphismp map (direct-product l) (direct-product m)))
           (permutationp (orders l) (orders m))))
```

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

[3] 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, doi:10.4204/EPTCS.359.10.

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