A modular formalisation of finite group theory

Georges Gonthier, Assia Mahboubi[†], Laurence Rideau[†],
Enrico Tassi[‡] and Laurent Théry[†]

Microsoft Research INRIA[†] University of Bologna[‡]
gonthier@microsoft.com tassi@cs.unibo.it[‡]
[Assia.Mahboubi|Laurent.Rideau|Laurent.Thery]@inria.fr[†]

Abstract. In this paper, we present a formalisation of elementary group theory done in Coq. This work is the first milestone of a long-term effort to formalise Feit-Thompson theorem. As our further developments will heavily rely on this initial base, we took special care to articulate it in the most compositional way.

1 Introduction

Recent works such as [2, 7, 8, 17] show that proof systems are getting sufficiently mature to formalise non-trivial mathematical theories. Group theory is a domain of mathematics where computer proofs could be of real added value. This domain was one of the first to publish *very long* proofs. The first and most famous example is the Feit-Thompson theorem. Its historical proof [6] is 255 pages long. That proof has later been simplified and re-published [4, 16], providing a better understanding of local parts of the proof. Yet its length remains unchanged, as well as its global architecture. Checking such a long proof with a computer would clearly increase the confidence in its correctness, and hopefully lead to a further step in the understanding of this proof. This paper addresses the ground work needed to start formalising this theorem.

There have been several attempts to formalise elementary group theory using a proof assistant. Most of them [1, 11, 21] stop at the Lagrange theorem. An exception is Kammüller and Paulson [12] who have formalised the first Sylow theorem. The originality of our work is that we do not use elementary group theory as a mere example but as a foundation for further formalisations. It is then crucial to us that our formalisation scales up. We have therefore worked out a new development, with a strong effort in proof engineering.

First of all, we reuse the SSReflect extension of Coq developed by Gonthier for his proof of the Four Colour theorem. This gives us a library and a proof language that is particularly well suited to the formalisation of finite groups. Second, we make use of many features of the Coq proof engine (notations, implicit arguments, coercions, canonical structures) to get more readable statements and tractable proofs.

The paper is organised as follows. In Section 2, we present the SSReflect extension and show how it is adequate to our needs. In Section 3, we comment

some of our choices in formalising objects such as groups, quotients and morphisms. Finally, in Section 4, we present some classic results of group theory that have already been formally proved in this setting.

2 Small scale reflection

The SSReflect extension [9] offers new syntax features for the proof shell and a bunch of libraries making use of *small scale reflection* in various respects. This layer above the standard Coq system provides a convenient framework for dealing with structures equipped with a decidable equality. In this section, we comment the fundamental definitions present in the library and how modularity is carried out throughout the development.

2.1 Proof shell

Proof scripts written with the SSReflect extension have a very different flavour than the ones developed using standard Coq tactics. We are not going to present the proof shell extensively but only describe some simple features, that, we believe, have a positive impact on productivity. A script is a linear structure composed of tactics. Each tactic ends with a period. An example of such a script is the following

```
move⇒ x a H; apply: etrans (cardUI _ _).
  case: (a x); last by rewrite /= card0 card1.
  by rewrite [_ + x]addnC.
by rewrite {1}mem_filter /setI.
```

All the frequent bookkeeping operations that consists in moving, splitting, generalising formulas from (or to) the context are regrouped in a single tactic **move**, making these operations more intuitive. For example, the fact that arguments of the **move** at the first line of the example of script are after the arrow indicates that the three arguments are the name to associate to three formulas to move from the conclusion to the context.

Good practise recommends to outline the underlying structure of the proof by indenting. To further structure scripts, SSReflect first proposes a tactical **by** to explicitly tag closing tactics. When replaying scripts, we then have the nice property that an error immediately occurs when a closing tactic fails to prove its subgoal. Second, when composing tactics, the two tacticals **first** and **last** let the user restrict the application of a tactic to only the first or the last subgoal generated by the previous command. It covers the frequents cases where a tactic generates two subgoals one of which can be easily discarded. In practice, these two tactics are so effective at increasing the linearity of our scripts that, in fact, it is very rare than more than two levels of indentation are needed.

Finally, the **rewrite** tactic in SSReflect comes with a concise syntax to accommodate in a single command all the possible combinations of conditional

rewriting, unfolding of definition, simplifying, rewriting selecting specific occurrences, rewriting selecting specific patterns, to name only some of them. Rewriting is then really convivial and contributes to a change of proof style more based on equational reasoning. In the standard library of Coq, the **rewrite** tactic is roughly used the same number of times than the tactic **apply**. In our development for group theory, **rewrite** is used three times more than **apply** — despite the fact that, on average, each SSReflect **rewrite** stands for three Coq **rewrite**s.

2.2 Views

The Coq system is based on an intuitionistic type theory, the Calculus of Inductive Constructions [19, 14]. There is a distinction between logical propositions and boolean values. On the one hand, logical propositions are objects of type Prop for which the excluded middle does not hold, i.e. the proposition $\forall$ P:Prop, P $\vee$ $\neg$P is not provable. On the other hand, bool is an inductive datatype with two constructors true and false, for which the term
fun b $\rightarrow$ **if** b returns (b || $\sim$b = true) **then** refl_equal true **else** refl_equal true
is a proof of $\forall$ b: bool, b || $\sim$b = true. This proof does a dependent case analysis on b and returns in each case a proof of true = true, the term (refl_equal true), thanks to the fact that boolean functions are computable.

When working in a decidable domain, the distinction between propositions and booleans does not make sense anymore. The small scale reflection proposes a generic mechanism to have the best of the two worlds and move freely from a propositional version of a decidable predicate to its boolean version. For this, booleans are injected into propositions using the coercion mechanism:

Coercion is_true (b: bool) := b = true.

Now, every time the Coq system expects a proposition but receives a boolean b, it will automatically coerce it into the proposition (is_true b), i.e the proposition b = true. Coercions are also omitted by the prettyprinter, so everything is completely transparent to the user. Then, the inductive predicate reflect is used to relate propositions and booleans

Inductive reflect (P: Prop): bool $\rightarrow$ Type :=
| Reflect_true : P $\Rightarrow$ reflect P true
| Reflect_false : $\neg$P $\Rightarrow$ reflect P false.

The statement (reflect b P) indicates that (is_true b) and P are two logically equivalent propositions. In the following, we use the notation b $\leftrightarrow$ P for (reflect b P). For instance, the following lemma:

Lemma andP: $\forall$ b_1 b_2, (b_1 $\wedge$ b_2) $\leftrightarrow$ (b_1 && b_2).

relates the boolean conjunction && and the logical one $\wedge$. Note that in andP, b_1 and b_2 are two boolean variables and the proposition b_1 $\wedge$ b_2 hides two coercions. The conjunction of b_1 and b_2 can then be viewed as $b_1 \wedge b_2$ or as b_1&& b_2. A naming convention in SSReflect is to postfix the name of view lemmas with P. For example, orP relates || and $\vee$, negP relates $\sim$ and $\neg$.

Views are integrated to the proof language. If we are to prove a goal of the form $(b_1 \wedge b_2) \to G$, the tactic **case** $\Rightarrow E_1\ E_2$ changes the goal to G adding to the context the two assumptions E_1: b_1 and E_2: b_2. If the goal is of the form $(b_1 \&\&\ b_2) \to G$ instead, we simply need to change the tactic to **case**/andP $\Rightarrow E_1\ E_2$ to perform the necessary intermediate change of view.

Suppose now that our goal is $b_1 \&\&\ b_2$. In order to split this goal into two subgoals, we use a combination of two tactics: **apply**/andP; **split**. The first tactic performs the change of view so that the second tactic can do the splitting. Note that if we happen to have in the context an assumption H: b_1, instead of performing the splitting, the tactic **rewrite** H /=, i.e., rewriting with H followed by a simplification, can directly be used to transform the goal $b_1 \&\&\ b_2$ into b_2.

Views also provide a convenient way to swap between several (logical) characterisations of the same (computational) definition, having a view lemma per interpretation. A trivial example is the ternary boolean conjunction. If we have a goal of the form $b_1 \&\& (b_2 \&\& b_3) \to G$, applying the tactic **case**/andP leads to the goal $b_1 \to b_2 \&\& b_3 \to G$. We can also define an alternative view with

Inductive and3 (P Q R: Prop): Prop := And3: P → Q → R → (and3 P Q R).

Lemma and3P: ∀ b_1 b_2 b_3, (and3 b_1 b_2 b_3) ↔ (b_1 && (b_2 && b_3)).

Now, the tactic **case**/and3P directly transforms the goal $b_1 \&\& (b_2 \&\& b_3) \to G$ into $b_1 \to b_2 \to b_3 \to G$.

2.3 Libraries

In our formalisation of finite groups, we reused the base libraries initially developed for the formal proof of the Four Colour theorem. They consist in a hierarchy of structures and a substantial toolbox to work with finite types. At the bottom of this hierarchy, the structure eqType deals with types with decidable equality.

```
Structure eqType : Type := EqType {
    sort :> Type;
  _ == _ : sort → sort → bool;
    eqP :  ∀ x y, (x = y) ↔ (x == y)
}.
```

The :> symbol declares sort as a coercion from an eqType to its carrier type. It is the standard technique to get subtyping, an object of type eqType can then be viewed as an object of type Type. In the type theory of Coq, the only relation we can rewrite with is the primitive (Leibniz) equality. When another equivalence relation is the intended notion of equality on a given type, the user usually needs to use the setoid workaround [3]. Unfortunately, setoid rewriting does not have the same power as primitive rewriting. An eqType structure not only assumes the existence of a decidable equality == but also eqP injects this equality into the Leibniz one, thus promoting it to a *rewritable* relation.

Any non parametric inductive type can be turned into an eqType choosing for == the function that checks structural equality. This is the case for booleans and

natural numbers for which a bool_eqType and nat_eqType are defined as *canonical structures*. Canonical structures are used when solving equations involving implicit arguments. Namely, if the type checker needs to infer an eqType structure on the type nat, it will automatically choose as a default choice the nat_eqType type. By enlarging the set of implicit arguments Coq can infer, canonical structures ease the handling of the hierarchy of structures.

A key property of eqType structures is that they enjoy proof-irrelevance for the equality proofs of their elements: every equality proof is convertible to a reflected boolean test.

Lemma eq_irrelevance: ∀ (d: eqType) (x y: d) (E: x = y) (E': x = y), E = E'.

An eqType structure only defines a domain, in which sets take their elements. Sets are then represented by their characteristic function

Definition set (d: eqType) := d → bool.

and defining set operations like ∪ and ∩ is done by providing the corresponding boolean functions.

The next step consists in building lists, elements of type seq d, whose elements belong to the parametric eqType structure d. The decidability of equality on d is needed when defining the basic operations on lists like membership ∈ and look-up index. Then, membership is used for defining a coercion from list to set, such that (l x) is automatically coerced into x ∈ l.

Lists are the cornerstone of the definition of finite types. A finType structure is composed of a list of elements of an eqType structure, each element being unique.

```
Structure finType : Type := FinType {
  sort :> eqType;
  enum : seq sort;
  enumP : ∀ x, count (set1 x) enum = 1
}.
```

where (set1 x) is the set that contains only x and (count f l) computes the number of elements y of the list l for which (f y) is true.

Finite sets are then sets taken in a finType domain. In the library, the basic operations are provided. For example, given A a finite set, (card A) represents the cardinality of A. All these operations come along with their basic properties. For example, we have:

```
Lemma cardUI : ∀ (d: finType) (A B: set d),
  card (A ∪ B) + card (A ∩ B) = card A + card B.
Lemma card_image : ∀ (d d': finType) (f: d → d') (A: set d),
      injective f ⇒ card (image f A) = card A.
```

3 The group library

This section is dedicated to the formalisation of elementary group theory. We justify our definitions and explain how they relate to each other.

3.1 Graphs of function and intensional sets

We use the notation f=$_1$g to indicate that two functions are extensionally equal, i.e the fact that $\forall$x, f x = g x holds. In Coq, f=$_1$g does not implies f = g. This makes equational reasoning with objects containing functions difficult in Coq without adding extra axioms. In our case, extra axioms are not needed. The functions we manipulate have finite domain so they can be finitely represented by their graph. Given d_1 a finite type and d_2 a type with decidable equality, a graph is defined as

```
Inductive fgraphType : Type :=
  Fgraph (val: seq d2) (fgraph_sizeP: size val = card d1): fgraphType.
```

It contains a list val of elements of d_2, the size of val being exactly the cardinal of d_1. Defining a function fgraph_of_fun that computes the graph associated to a function is straightforward. Conversely, a conversion fun_of_fgraph is defined to let the user manipulate graphs as standard functions. With graphs as functions, it is possible to prove functional extensionality

```
Lemma fgraphP : ∀ (f g : fgraphType d1 d2), f =1 g ⇔ f = g.
```

Note that on the left-hand side of the equivalence, f =$_1$g is automatically coerced into (fun_of_graph f) =$_1$(fun_of_graph g). In order to make graphs a proper substitute to functions, we need to equip them with the same operations that the ones proposed for functions. For example, (setType d) corresponds to (set d). We call elements of (setType d) *intensional* sets by opposition to the sets defined by their characteristic function. The notation {x, f x} is used to define the intensional set whose characteristic function is f and (iimage f A) corresponds to the intensional set of the image of A by f.

Graphs are used to build some useful datastructures. For example, homogeneous tuples, i.e. sequences of elements of type K of fixed length n, are implemented as graphs with domain (ordinal n), the finite type {0, 1, 2,..., n−1}, and co-domain K. With this representation, the n-th element of a p-tuple t can be obtained applying t to n, as soon as n lies in the the domain of t. Also, permutations are defined as function graphs with identical domain and co-domain, the val list of which does not contain any duplicate.

3.2 Groups

In the same way than eqType structures were introduced before defining sets, we introduce a notion of (finite) *group domain* which is distinct from the one of groups. It is modelled by a finGroupType record structure

```
Structure finGroupType : Type := FinGroupType {
  element :> finType;
        1 : element;
     _^-1 : element → element;
  _ * _ : element → element → element;
  unitP : ∀ x, 1 * x = x;
```

```
  invP : ∀ x, x^-1 * x = 1;
  mulP : ∀ x1 x2 x3, x1 * (x2 * x3) = (x1 * x2) * x3
}.
```

It contains a carrier, a composition law and an inverse function, a unit element and the usual properties of these operations. Its first field is declared as a coercion to the carrier of the group domain.

In the group library, a first category of lemmas is composed of properties that are valid on the whole group domain. For example:

```
Lemma invg_mul : ∀ x1 x2, (x2 * x1)^-1 = x1^-1 * x2^-1.
```

Also, we can already define operations on arbitrary sets of a group domain. If A is such a set, we can define for instance:

```
Definition  x ^ y  := y^-1 * x * y.
Definition A :* x  := {y, y * x^-1 ∈ A}. (* right cosets *)
Definition A :^ x  := {y, y ^ x^-1 ∈ A}. (* conjugate *)
Definition normaliser A := {x, (A :^ x) ⊂ A}.
```

Some definitions may look less intuitive at first sight since we try as much as possible to define them as boolean predicates. For example, the set of point-wise products of two sets is defined as:

```
Definition A :*: B := {xy, ~(disjoint {y, xy ∈ (A :* y)} B)}
```

A *view* lemma gives the natural characterisation of this object:

```
Lemma smulgP : ∀ A B z, (∃ x y, x ∈ A & y ∈ B & z = x * y) ↔ (z ∈ A :*: B).
```

Lemmas like smulgP belongs to the second category of lemmas composed of the properties of these operations requiring only group domain *sets*.

Finally, a *group* is defined as a boolean predicate, satisfied by sets of a given group domain that contain the unit and are stable under product.

```
Definition group_set A := 1 ∈ A && (A :*: A) ⊂ A.
```

It is very convenient to give the possibility of attaching in a canonical way the proof that a set has a group structure. This is why groups are declared as structures:

```
Structure group(elt : finGroupType) : Type := Group {
  set_of_group :> setType elt;
  set_of_groupP : group_set set_of_group
}.
```

The first argument of this structure is a *set*, giving the carrier of the group. Notice that we do *not* define one type per group but one type per group domain, which avoids having unnecessary injections everywhere in the development.

Finally, the last category of lemmas in the library is composed of group properties. For example, given a group H, we have the following property:

```
Lemma groupMl : ∀ x y, x ∈ H ⇒ (x * y) ∈ H = y ∈ H.
```

In the above statement, the equality stands for Coq standard equality between boolean values, since membership of H is a boolean predicate.

We declare a canonical group structure for the usual group constructions so that they can be displayed as their set carrier but still benefit from an automatically inferred proof of group structure when needed. For example, such canonical structure is defined for the intersection of two groups H and K that share the group domain elt:

Lemma group_setI : group_set (H ∩ K).
Canonical Structure setI_group := Group group_setI.

where, as in the previous section, ∩ stands for the *set* intersection operation. Given a group domain elt and two groups H and K, the stability of the group law for the intersection is proved in the following way:

Lemma setI_stable : ∀ x y, x ∈ (H ∩ K) ⇒ y ∈ (H ∩ K) ⇒ (x * y) ∈ (H ∩ K).
Proof. **by move**⇒ x y H1 H2; **rewrite** groupMl. **Qed**.

The group structure on the H ∩ K carrier is automatically inferred from the canonical structure declaration and the **by** closing command uses the H1 and H2 assumptions to close two trivial generated goals.

This two-level definition of groups, involving group domain types and groups as first order citizens equipped with canonical structures, plays an important role in doing proofs. Type inference is then used to perform proof inference from the database of registered canonical structures.

3.3 Quotients

Typically, every local section of our development assumes once and for all the existence of one group domain elt to then manipulate different groups of this domain. Nevertheless, there are situations where it is necessary to build new finGroupType structures. This is the case for example for *quotients*. Let H and K be two groups in the same group universe, the quotient K/H is a group under the condition that H is *normal* in K. Of course, we could create a new group domain for each quotient, but we can be slightly smarter noticing that given a group H, all the quotients of the form K/H share the same group law, and the same unit. The idea is then to have all the quotients groups K/H in a group domain ./H. The largest possible quotient is N(H)/H, where N(H) is the normaliser of H and all the other quotients are subsets of this one.

In our formalisation, normality is defined as:

Definition H ◁ K := (H ⊂ K) && (K ⊂ (normaliser H)).

If H◁K, H-left cosets and H-right cosets coincide for every element of K. Hence, they are just called *cosets*. Once again, we carefully stick to first order predicates to take as much benefit as possible from the canonical structure mechanism. If necessary, side conditions are embedded inside definitions by the mean of boolean tests. Like this, we avoid having to add pre-conditions in the properties of these predicates to insure well-formedness. The definition of cosets makes no restriction on its arguments:

Definition coset (A : setType elt) (x : elt) :=
  if (x ∈ (normaliser A)) **then** A :* x **else** A.

The set of cosets of an arbitrary set A is the image of the whole group domain by the coset operation. Here we define the associated sigma type:

Definition cosets (A : setType elt):= iimage (coset A) elt.
Definition cosetType (A : setType elt):= eq_sig (cosets A).

where eq_sig builds the sigma type associated to a set. This cosetType type can be equipped with canonical structures of eqType and finType and elements of this type are intentional sets.
The quotient of two groups of the same group domain can *always* be defined:

Definition A/B := iimage (coset_of B) A.

where coset_of : elt → (cosetType A) injects the value of (coset A x) in (cosetType A). Thanks to the internal boolean test in coset, A/B defines in fact $[A \cap N(B)]/B$.

When H is equipped with a group structure, we define group operations on (cosetType H) thanks to the following properties:

Lemma cosets_unit : H ∈ (cosets H).
Lemma cosets_mul : ∀ Hx Hy : cosetType H, (Hx :*: Hy) ∈ (cosets H).
Lemma cosets_inv : ∀ Hx : cosetType H, (Hx :$^{-1}$) ∈ (cosets H).

where A :$^{-1}$ denotes the image of a set A by the inverse operation. Group properties are provable for these operations: we can define a canonical structure of group domain on cosetType, depending on an arbitrary group object. Canonical structures of *groups*, in this group domain, are defined for every quotient of two group structures. A key point in the readability of statements involving quotients is that the ./. notation is usable because it refers to a definition independent of proofs; the type inference mechanism will automatically find an associated group structure for this set when it exists.

Defining quotients has also been a place where we had to rework our formalisation substantially using intensional sets instead of sets defined by their characteristic function. In the library of finite group quotients, there are two kinds of general results. The first one states *equalities* between quotients, like the theorems about the kernel of quotient morphism. The second, often heavily relying on properties of the first kind, builds isomorphisms between different groups, i.e. groups having distinct carriers (and hence operations). For example, this is the case for the so-called three fundamental isomorphism theorems. The initial version of the quotients was using sets defined by their characteristic function. Having sets for which function extensionality does not hold had forced us to use setoid. For theorems with types depending on setoid arguments, especially the ones stating equalities, we had to add one extensional equality condition per occurrence of such a dependant type in the statement of the theorem in order to make these theorems usable. The situation was even worse since, in order to apply one of these theorems. the user had to provide specific lemmas, proved before-hand, for each equality proof. This was clearly unacceptable if quotients

were to be used in further formalisations. Using intensional sets has simplified everything.

3.4 Group Morphisms

Group morphisms are functions between two group domains, which comply with the group laws of their domain and codomain. Since we do not create one type per group, the notion of morphism is parametrised by a group on which morphism properties hold. The fundamental property of group morphisms is that they preserve group structures under image and pre-image.

To avoid having to use technical lemmas about the restriction of morphism domains, we want the image and preimage of groups by morphism to have a *canonical* structure of group. Thus, the values of a given function alone should be enough to determine the largest group on which this function may be seen as a morphism.

We have embedded the domain of a morphism inside its computational definition by giving a default unit value outside the group where the morphism properties are supposed to hold. Now, the problem is to compute back the domain of a morphism candidate from its values, identifying the kernel among the set of elements mapped to the unit:

Definition ker (f: elt → elt') := {x:elt ⊂ {y: elt, f (x ∗ y) == f y}}.

which can be equipped with a canonical group structure. Morphism domains are defined as:

Definition mdom (f: elt → elt') := ker ∪ {x, f x != 1}.

Morphisms are defined by the following structure:

```
Structure morphism : Type := Morphism {
  mfun :> elt → elt';
  group_set_mdom : group_set (mdom mfun);
  morphM : ∀ x y,
   (mfun x) ∈ mdom ⇒ (mfun y) ∈ mdom ⇒ mfun (x ∗ y) = mfun x ∗ mfun y
}.
```

An isomorphism is a morphism having a trivial kernel. Restricting a morphism is simply done by giving the default unit value outside its intended domain. This operation is a canonical morphism construction. Morphisms and quotients are involved in the universal property of morphism factorisation. For any function between group domains, we define a quotient function by:

```
Definition fquo H (f : elt → elt'):=
  if H ⊂ (ker f) then fun (Hx : cosetType H) ⇒ f (repr Hx)
  else fun (Hx : cosetType H) ⇒ 1.
```

where repr picks a representative in any set of a finGroupType. Given any morphism, its quotient function defines an isomorphism between the quotient of its domain by its kernel and the image of the initial morphism.

This definition of morphism has been motivated by the formal proofs of the three fundamental isomorphism theorems. The goal was to eliminate any proof dependency which cannot be resolved by the type inference system with the help of canonical structures. The result is that statements are much more readable and formal proofs much easier. For instance, the third fundamental isomorphism theorem follows directly from the three lemmas below, because the function f3 is *canonically* a morphism.

Hypothesis sHK : H ⊂ K.
Hypothesis nHG : H ◁ G.
Hypothesis nKG : K ◁ G.

Let f3 := (fquo (fquo (coset K)).

Lemma mdom_f3 : mdom f3 ⊂ (G / H) / (K / H).
Lemma im_f3 : iimage f3 = G / K.
Lemma f3_ker : (ker f3) = {1}.

4 Standard theorems of group theory

In order to evaluate how practical our definitions of groups, cosets and quotients were, we have started formalising some standard results of group theory. In this section, we present three of them: Sylow theorems, Frobenius lemma and Cauchy-Frobenius lemma. Sylow theorems are central in group theory. Frobenius lemma gives a nice property of the elements of a group of a given order. Finally Cauchy-Frobenius lemma, also called Burnside counting lemma, applies directly to enumeration problems. Our main source of inspiration for these proofs was some lecture notes on group theory by Constantine [5].

4.1 Sylow theorems

The first Sylow theorem states the existence of a subgroup `H` of `K` of cardinal $\mathtt{p}^n$, for every prime `p` such that $\mathtt{card(K)} = \mathtt{p}^n s$ and p does not divide `s`. Its formal statement is the following

Definition sylow K p H := subgroup H K && card H == p ^ (logn p (card K)).

Theorem sylow1: ∀ K p, ∃ H, sylow K p H.

The first definition captures the property of `H` being a `p`-Sylow subgroup of `K`. The function `logn` computes, if `p` is prime, the maximum value of i such that $\mathtt{p}^i$ divides the cardinality of `K`, if `p` is not prime it returns 0. This theorem has already been formalised by Kammüller and Paulson [12]. They have followed the standard proof due to Wielandt [20]. Our proof is slightly different and intensively uses group actions on sets. Given a group domain G and a finite type S, actions are defined by the following structure

```
Structure action : Type := Action {
  act_f :> S → G →S;
  act_1 : ∀x, act_f x 1 = x;
  act_morph : ∀(x y : G) z, act_f z (x ∗ y) = act_f (act_f z x) y
}.
```

Note that we take advantage of our finite setting to replace the usual bijectivity of the action by the simpler property that acting with the neuter element is the identity.

A complete account of our proof is given in [18]. The proof works by induction on n showing that there exists a subgroup of order $\mathtt{p}^i$ for all $0 < i \leq n$. The base case is Cauchy theorem. It states the existence of an element of order $\mathtt{p}$ where $\mathtt{p}$ is a prime divisor of the cardinality of the group $\mathtt{K}$. To prove it, we use a simpler argument that the one in [12] where a combinatorial argument based on some properties of the binomial is used. We first build the set U such that $U = \{(k_1, \ldots, k_{\mathtt{p}}) \mid k_i \in \mathtt{K} \text{ and } \prod_{i=1}^{i=\mathtt{p}} k_i = 1\}$. We have that $card(U) = card(\mathtt{K})^{\mathtt{p}-1}$. We then define the action of the additive group $\mathbb{Z}/\mathtt{p}\mathbb{Z}$ that acts on U as

$$n \quad \longmapsto \quad (k_1, \ldots, k_{\mathtt{p}}) \mapsto (k_{n \bmod \mathtt{p}+1}, \ldots, k_{(n+\mathtt{p}-1) \bmod \mathtt{p}+1})$$

Note that defining this action is straightforward since p-tuples are graphs of function which domain is (ordinal p).

Now, we consider the set S_0 of the elements of U whose orbits by the action are a singleton. S_0 is composed of the elements $(k, \ldots, k)$ such that $k \in \mathtt{K}$ and $k^p = 1$. A consequence of the class equation tells us that $\mathtt{p}$ divides the cardinal of S_0. As S_0 is non-empty ($(1, \ldots, 1)$ belongs to S_0), there exists at least one $k \neq 1$, such that $(k, \ldots, k)$ belongs to S_0. The order of k is then $\mathtt{p}$.

In a similar way, in the inductive case, we suppose that there is a subgroup $\mathtt{H}$ of order $\mathtt{p}^i$, we consider $N_{\mathtt{K}}(\mathtt{H})/\mathtt{H}$ the quotient of the normaliser of $\mathtt{H}$ in $\mathtt{K}$ by $\mathtt{H}$. We act with $\mathtt{H}$ on the left cosets of $\mathtt{H}$ by left translation:

$$g \quad \longmapsto \quad h\mathtt{H} \mapsto (gh)\mathtt{H}$$

and consider the set S_0 of the left coset of $\mathtt{H}$ whose orbits by the action are a singleton. The elements of S_0 are exactly the elements of $N_{\mathtt{K}}(\mathtt{H})/\mathtt{H}$. Again, applying the class equation, we can deduce that $\mathtt{p}$ divides the cardinal of S_0 so there exists an element k of order $\mathtt{p}$ in S_0 by Cauchy theorem. If we consider $\mathtt{H'}$ the pre-image by the quotient operation of the cyclic group generated by k, its cardinality is $\mathtt{p}^{i+1}$.

We have also formalised the second and third Sylow theorems. The second theorem states that any two $\mathtt{p}$-Sylow subgroups H_1 and H_2 are conjugate. This is proved acting with H_1 on the left coset of H_2. The third theorem states that the number of $\mathtt{p}$-Sylow subgroups divides the cardinality of $\mathtt{K}$ and is equal to 1 modulo p. The third theorem is proved by acting by conjugation on the sets of all $\mathtt{p}$-Sylow subgroups.

4.2 Frobenius lemma

Given an element a of a group G, (cyclic a) builds the cyclic group generated by a. When proving properties of cyclic groups, we use the characteristic property of the cyclic function.

Lemma cyclicP: ∀ a b, reflect (∃ n, a^ n = b) (cyclic a b).

The order of an element is then defined as the cardinality of its associated cyclic group. Frobenius lemma states that given a number n that divides the cardinality of a group K, the number of elements whose order divides n is a multiple of n. In our formalisation, this gives

Theorem frobenius: ∀ K n, n | (card K) → n | (card {z:K, (orderg z) | n}).

The proof is rather technical and has intensively tested our library on cyclic groups. For example, as we are counting the number of elements of a given order, we need to know the number of generators of a cyclic group. This is given by a theorem of our library

Lemma phi_gen: ∀ a,phi (orderg a) = card (generator (cyclic a)).

where phi is the Euler function.

4.3 The Cauchy-Frobenius lemma

Let G a group acting on a set S. For each g in G, let F_g be the set of elements in S fixed by g, and t the number of orbits of G on S, then t in equal to the average number of points left fixed by each element of G:

$$t = \frac{1}{|G|} \sum_{g \in G} |F_g|$$

To prove this lemma, we consider B, subset of the cartesian product $G \times S$ containing the pairs (g, x) such that $g(x) = x$. We use two ways to evaluate the cardinality of B, first by fixing the first component: $|B| = \sum_{g \in G} |F_g|$, then by fixing the second component: $|B| = \sum_{x \in S} |G_x|$ where G_x is the stabiliser of x in G. Then, when sorting the right hand-side of the second equality by orbits we obtain that $|B| = |Gx_1||G_{x_1}| + |Gx_2||G_{x_2}| + \cdots + |Gx_t||G_{x_t}|$ the x_i being representatives of the orbit Gx_i. Applying the Lagrange theorem on the stabiliser of x_i in G (the subgroup G_{x_i}), we obtain that for each orbit: $|Gx_i||G_{x_i}| = |G|$ and we deduce that $|B| = t|G| = \sum_{g \in G} |F_g|$.

This lemma is a particular case of the powerful Pólya method, but it already has significant applications in combinatorial counting problems. To illustrate this, we have formally shown that there are 55 distinct ways of colouring with 4 colours the vertices of a square up to isometry. This is done by instantiating a more general theorem that tells that the number of ways of colouring with n colours is $(n^4 + 2n^3 + 3n^2 + 2n)/8$. This last theorem is a direct application of the Cauchy-Frobenius theorem. The encoding of the problem is the following:

Definition square := ordinal 4.
Definition colour := ordinal n.
Definition colouring := fgraphType square colour.

Vertices are represented by the set $\{0, 1, 2, 3\}$, colours by the set $\{0, 1, \ldots, n-1\}$ and colouring by functions from vertices to colours. The set of isometries is a subset of the permutations of `square` that preserve the geometry of the square. In our case, we use the characteristic condition that *the images of two opposite vertices remain opposite.*

Definition isometry := {p : perm square, ∀i, p (opp i) = opp (p i)}.

where perm square the permutation group and opp the function that returns the opposite of a vertex. We get that the isometries is a subgroup of the permutations, since the property of conserving opposite vertices is stable by composition and the identity obviously preserve opposite vertices.

Given p an isometry, acting with p is defined as the function that given a colouring c returns the colouring $i \mapsto c(p(i))$. Each set of identical coloured squares corresponds to an orbit of this action. To apply Cauchy-Frobenius, we first need to give an extensional definition of the isometries, i.e. there are 8 isometries: the identity, the 3 rotations of $\pi/2$, π and $3\pi/2$, the vertical symmetry, the horizontal symmetry and the 2 symmetries about the diagonals. Second, we have to count the elements left fixed by each of the isometry.

The proofs of three theorems presented in this section manipulate many of the base concepts defined in our formalisation. They have been particularly important to gave us feed-back on how practical our definitions were.

5 Conclusion

To our knowledge, what is presented in this paper is already one of the most complete formalisation of finite group theory. We almost cover all the material that can be found in an introductory course on group theory. Very few standard results like the simplicity of the alternating group are still missing, but should be formalised very soon. The only similar effort but in set theory can be found in the Mizar system [13]. Theorems like the ones presented in Section 4 are missing from the Mizar formalisation.

Getting the definitions right is one of the most difficult aspect of formalising mathematics. The problem is not much in capturing the semantics of each individual construct but rather in having all the concepts working together well. Group theory has been no exception in that respect. We had lots of try and go before converging to the definitions presented in this paper. The fact that we were able to get results like the ones presented in Section 4 relatively easily makes us confident that our base is robust enough to proceed to further formalisations.

Using SSReflect has been a key aspect to our formal development. It gives us a very effective way of doing proofs inside the Coq system. Using decidable types and relying heavily on rewriting for our proofs gives a 'classical' flavour to our development that is more familiar to what can be found in provers like

Isabelle [15] or Hol [10] than what is usually done in Coq. An indication of the conciseness of our proof scripts is given by the following figure. The standard library of Coq contains 7000 objects (definitions + theorems) for 93000 lines of code, this makes a ratio of 13 lines per object. The base library of SSReflect plus our library for groups contains 1980 objects for 14400 lines, this makes a ratio of 7 lines per object.

References

1. Rob Arthan. Some group theory. Available at `http://www.lemma-one.com/ProofPower/examples/wrk068.pdf`.
2. Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A Formally Verified Proof of the Prime Number Theorem. *ACM Transactions on Computational Logic*, To appear.
3. G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, March 2003.
4. Helmut Bender and Georges Glauberman. *Local analysis for the Odd Order Theorem*. Number 188 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1994.
5. Gregory M. Constantine. Group Theory. Available at `http://www.pitt.edu/~gmc/algsyl.html`.
6. Walter Feit and John G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3):775–1029, 1963.
7. Herman Geuvers, Freek Wiedijk, and Jan Zwanenburg. A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals. In *Types for Proofs and Programs, TYPES 2000 International Workshop, Selected Papers*, volume 2277 of *LNCS*, pages 96–111, 2002.
8. Georges Gonthier. A computer-checked proof of the four-colour theorem. Available at `http://research.microsoft.com/~gonthier/4colproof.pdf`.
9. Georges Gonthier. Notations of the four colour thorem proof. Available at `http://research.microsoft.com/~gonthier/4colnotations.pdf`.
10. Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL : a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.
11. Elsa Gunter. Doing Algebra in Simple Type Theory. Technical Report MS-CIS-89-38, University of Pennsylvania, 1989.
12. Florian Kammüller and Lawrence C. Paulson. A Formal Proof of Sylow's Theorem. *Journal of Automating Reasoning*, 23(3-4):235–264, 1999.
13. The Mizar Home Page. `http://www.mizar.org/`.
14. C. Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.
15. Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994.
16. Thomas Peterfalvi. *Character Theory for the Odd Order Theorem*. Number 272 in London Mathematical Society Lecture Note Series. Cambridge University Press, 2000.
17. The Flyspeck Project. `http://www.math.pitt.edu/~thales/flyspeck/`.
18. Laurence Rideau and Laurent Théry. Formalising Sylow's theorems in Coq. Technical Report 0327, INRIA, 2006.

19. Benjamin Werner. *Une théorie des Constructions Inductives*. PhD thesis, Paris 7, 1994.
20. Helmut Wielandt. Ein beweis für die Existenz der Sylowgruppen. *Archiv der Mathematik*, 10:401–402, 1959.
21. Yuan Yu. Computer Proofs in Group Theory. *J. Autom. Reasoning*, 6(3):251–286, 1990.