A Modular Formalisation of Finite Group Theory*

Georges Gonthier[1], Assia Mahboubi[2], Laurence Rideau[3], Enrico Tassi[4], and Laurent Théry[3]

[1] Microsoft Research
[2] Microsoft Research - Inria Joint Center
[3] INRIA
[4] 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 the 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 [2,8,9,19] 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, which states that every odd order group is solvable. Its historical proof [7] is 255 pages long. That proof has later been simplified and re-published [5,18], 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 [1,12,23] stop at the Lagrange theorem. An exception is Kammüller and Paulson [13] 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 for our formalisation to scale up. We have therefore worked out a new development, with a strong effort in proof engineering.

First, we reuse the SSReflect extension of the Coq system [14] developed by Gonthier for his proof of the Four Colour theorem. This gives us a library

* This work was supported by the Microsoft Research - Inria Joint Center.

K. Schneider and J. Brandt (Eds.): TPHOLs 2007, LNCS 4732, pp. 86–101, 2007.

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 on 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 [10] offers new syntax features for the proof shell and a set 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 on 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 the most 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 formulae from (or to) the context are regrouped in a single tactic **move**. For example, the tactic **move⇒** x a H on the first line moves (introduces) two constants x and a and an assumption H from the goal to the context.

It is recommended practise to use indentation to display the control structure of a proof. To further structure scripts, SSReflect supplies a **by** tactical to explicitly close off tactics. When replaying scripts, we then have the nice property that an error immediately occurs when a closed tactic fails to prove its subgoal. 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. This covers the frequent cases where a tactic generates two subgoals one of which can be easily disposed of. In practice, these two tacticals are so effective at increasing the linearity of our scripts that, in fact, we very rarely need more than two levels of indentation.

Finally, the **rewrite** tactic in SSReflect provides a concise syntax that allows a single command to perform a combination of conditional rewriting, folding and unfolding of definitions, and simplification, on selected patterns and even specific occurrences. This makes rewriting much more user-friendly and contributes to shift the proof style towards equational reasoning. In the standard library of Coq, the **rewrite** tactic is roughly used the same number of times as the **apply** tactic. 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 [21,16]. In such a formalism, there is a distinction between logical propositions and boolean values.

On the one hand, logical propositions are objects of *sort* Prop which is the carrier of intuitionistic reasoning. Logical connectives in Prop are *types*, which give precise information on the structure of their proofs; this information is automatically exploited by Coq tactics. For example, Coq knows that a proof of $A \vee B$ is either a proof of A or a proof of B. The tactics **left** and **right** change the goal $A \vee B$ to A and B, respectively; dually, the tactic **case** reduces the goal $A \vee B \Rightarrow G$ to two subgoals $A \Rightarrow G$ and $B \Rightarrow G$.

On the other hand, bool is an inductive *datatype* with two constructors true and false. Logical connectives on bool are computable *functions*, defined by their truth tables, using case analysis:

Definition $(b_1 \,||\, b_2) :=$ **if** b_1 **then** true **else** b_2.

Properties of such connectives are also established using case analysis: the tactic **by case**: b solves the goal $b \,||\, \sim b = \text{true}$ by replacing b first by true and then by false; in either case, the resulting subgoal reduces by computation to the trivial true = true.

Thus, Prop and bool are truly complementary: the former supports robust natural deduction, the latter allows brute-force evaluation. SSReflect supplies 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.

First, booleans are injected into propositions using the coercion mechanism:

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

This allows any boolean formula b to be used in a context where Coq would expect a proposition, e.g., after **Lemma** ... :. It is then interpreted as (is_true b), i.e., the proposition b = true. Coercions are elided by the prettyprinter, so they are essentially transparent to the user. Then, the inductive predicate reflect is used to relate propositions and booleans

Inductive reflect (P: Prop): bool → Type :=
| Reflect_true: P ⇒ reflect P true
| Reflect_false : ¬P ⇒ reflect P false.

The statement (reflect P b) asserts that (is_true b) and P are logically equivalent propositions. In the following, we use the notation P $\leftrightarrow$ b for (reflect P b).

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 && to 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 $(b_1 \wedge b_2) \Rightarrow G$, 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. To handle a goal of the form $(b_1\ \&\&\ b_2) \Rightarrow G$, we can simply annotate the tactic to specify an intermediate change of view: **case**/andP $\Rightarrow E_1\ E_2$.

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

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

Lemma and3P: $\forall\, b_1\ b_2\ b_3$, (and3 $b_1\ b_2\ b_3$) $\leftrightarrow$ ($b_1\ \&\&\ (b_2\ \&\&\ b_3)$).

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

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. These libraries build a hierarchy of structures using nested dependent record types. This technique is standard in type-theoretic formalisations of abstract algebra [3]. 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. If d is an object of type eqType, then an object x can have the type x : d, thanks to the sort coercion. The complete judgement is in fact x : sort d. Moreover, if x and y are of type sort d, the term x ≡ y is understood as the projection of d on its second field, applied to x and y. The implicit d parameter is both inferred and hidden from the user.

In the type theory of Coq, the only relation we can freely 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 [4]. Unfortunately, setoid rewriting does not have the same power as primitive rewriting. An eqType structure provides not only a computable equality ≡ but also a proof eqP that this equality reflects the Leibniz one; the eqP view thus promotes ≡ to a *rewritable* relation.

A non parametric inductive type can usually be turned into an eqType by choosing for ≡ the function that checks structural equality. This is the case for booleans and natural numbers for which we define the two *canonical structures* bool_eqType and nat_eqType. Canonical structures are used when solving equations involving implicit arguments. Namely, if the type checker needs to infer an eqType structure with sort nat, it will automatically choose the nat_eqType structure. By enlarging the set of implicit arguments Coq can infer, canonical structures make hierarchical structures widely applicable.

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 provides an enumeration of its sort: a sequence in which each element of type sort occurs exactly once. Note that sort must also have an eqType structure.

```
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 whose domain has a finType structure. The library provides many set operations. 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 Intentional Sets

We use the notation f=$_1$g to indicate that two functions are extensionally equal, i.e., the fact that ∀x, f x = g x holds. In Coq, f=$_1$g does not imply 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 domains 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 d_2) (fgraph_sizeP: size val = card d_1): fgraphType.

It contains a list val of elements of d_2, the size of val being exactly the cardinality of d_1. Defining a function fgraph_of_fun that computes the graph associated to a function is straightforward. Conversely, a coercion fun_of_fgraph lets the user use graphs as standard functions. With graphs as functions, it is possible to prove functional extensionality:

Lemma fgraphP : ∀(f g : fgraphType d_1 d_2), f =$_1$ g ⇔ f = g.

Note that here f =$_1$g is understood as (fun_of_graph f) =$_1$ (fun_of_graph g).

The special case of graphs with domain d and codomain bool_eqType is denoted (setType d). We call elements of (setType d) *intentional* sets by opposition to the sets in (set d), defined by their characteristic function. We equip intentional sets with the same operations as extensional sets. {x,E(x)} denotes the intentional set whose characteristic function is (fun x → E(x)), and (f @ A) denotes the intentional set of the image of A by f.

Graphs are used to build some useful data-structures. 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 p-th element of a n-tuple t can

be obtained applying t to p, as soon as p lies in the the domain of t. Also, permutations are defined as function graphs with identical domain and co-domain, whose val list does not contain any duplicate.

3.2 Groups

In the same way that 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, like it was the case in section 2.3. In particular, we can again define convenient global notations like $*$ or $^{-1}$ for the projections of this record type.

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}.
```

Providing a boolean predicate sometimes requires a little effort, e.g., in the definition of the *point-wise* product of two sets:

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

The corresponding *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 belong to 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, of type:

```
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. As a consequence, most of the theorems are stated like in a set-theoretic framework (see section 3.4 for some examples). Type inference is then used to perform the proof inference, from the database of registered canonical structures. This mechanism is used extensively throughout the development, and allows us in particular to state and use theorems in a way that follows standard mathematical practice.

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.

In our finite setting, the largest possible quotient exists and 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*.

The set of cosets of an arbitrary set A is the image of the normaliser of A by the rcoset operation. Here we define the associated sigma type:

Definition coset_set (A : setType elt):= (rcoset A) @ (normaliser A).
Definition coset (A : setType elt):= eq_sig (coset_set A).

where eq_sig builds the sigma type associated to a set. This coset type can be equipped with eqType and finType canonical structures; elements of this type are intentional sets.

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

Lemma cosets_unit : H ∈ (cosets H).
Lemma cosets_mul : ∀ Hx Hy : coset H, (Hx :*: Hy) ∈ (cosets H).
Lemma cosets_inv : ∀ Hx : coset 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 coset depending on an arbitrary group object.

The quotient of two groups of the same group domain can *always* be defined:

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

where coset_of : elt →(coset A) maps elements of the normaliser A to their coset, and the other ones to the coset unit. Hence A / B defines in fact $N_A(B)B/B$.

Every quotient G / H of two group structures is equipped with a canonical structure of *group* of the coset H group domain.

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 been a place where we had to rework our formalisation substantially using intentional sets instead of sets defined by their characteristic function.

A significant part of the library of finite group quotients deals with constructions of group isomorphisms. The first important results to establish are the so-called three fundamental isomorphism theorems.

Two isomorphic groups do not necessarily share the same group domain. For example, two quotients by two different groups will have distinct (dependent) types. Having sets for which function extensionality does not hold had forced us to use setoids. 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 dependent type in the statement of the theorem in order to make these theorems usable. Worse, 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 intentional sets simplified everything.

3.4 Group Morphisms

Group morphisms are functions between two group domains, compatible with the group laws of their domain and co-domain. Their properties may not hold on the whole group domain, but only on a certain group of this domain. The notion of morphism is hence a local one.

We avoid the numerous difficulties introduced by formalising partial functions in type theory by embedding the domain of a morphism inside its computational definition. Any morphism candidate takes a default unit value outside the group where the morphism properties are supposed to hold. Now, we can 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: $\mathrm{elt}_1 \to \mathrm{elt}_2$) := {x, elt_1 ⊂ {y, f (x ∗ y) ≡ f y}}.

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

Definition mdom (f: $\mathrm{elt}_1 \to \mathrm{elt}_2$) := ker f ∪ {x, f x ≠ 1}.

As a group is defined as a set plus certain properties satisfied by this set, a morphism is defined as a function, together with the properties making this function a morphism.

```
Structure morphism : Type := Morphism {
  mfun :> elt_1 → elt_2;
  group_set_mdom : group_set (mdom mfun);
  morphM : ∀ x y, x ∈ (mdom f) ⇒ y ∈ (mdom f) ⇒
                    mfun (x ∗ y) = (mfun x) ∗ (mfun y)
}.
```

The domain mdom f of a given morphism f is hence canonically equipped with an obvious group structure.

Thanks to the use of a default unit value, the (functional) composition of two morphisms is canonically equipped with a morphism structure. Other standard

constructions have a canonical morphism structure, like coset_of G as soon as G is a group.

Morphisms and quotients are involved in the universal property of morphism factorisation, also called first isomorphism theorem in the literature. We carefully stick to first order predicates to take as much benefit as possible from the canonical structure mechanism. If necessary, we embed side conditions inside definitions with a boolean test. In this way we avoid having to add pre-conditions in the properties of these predicates to ensure well-formedness. For any function between group domains, we define a quotient function by:

Definition mquo(f : $\text{elt}_1 \to \text{elt}_2$)(A : setType elt_1)(Ax : coset A):=
if A ⊂ (ker f) **then** f (repr Ax) **else** 1.

where repr Ax picks a representative of a coset Ax. 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.

Theorem first_isomorphism : ∀ H : group elt_1, ∀ f : morphism elt_1 elt_2,
H ⊂ (dom f) ⇒ isog (H / ($\ker_H$ f)) (f @ H).

An isomorphism between A and B is a morphism having a trivial kernel and mapping A to B. This localised version of the theorem builds an isomorphism between $\ker_H$f, the intersection of a group H with ker f, and the image f @ H of H.

This definition of morphisms is crafted to eliminate any proof dependency which cannot be resolved by the type inference system with the help of canonical structures. A pointed out in section 3.2, statements are much more readable and formal proofs much easier.

To convince the reader of the efficiency of these canonical structure definitions, we provide hereafter a verbatim copy of the statements of the isomorphism theorems we have formally proved.

```
Theorem second_isomorphism : forall G H : group elt,
  subset G (normaliser H) -> isog (G / (G :&: H)) (G / H).

Lemma quotient_mulg : forall G H : group elt, (G :*: H) / H = G / H.

Theorem third_isomorphism  : forall G H K : group elt,
  H <| G -> K <| G -> subset K H -> isog ((G / K) / (H / K)) (G / H).
```

Here `(G :&: H)` is ASCII for $G \cap H$. The `quotient_mulg` identity follows from our design choices for the formalisation of quotients. Rewriting with `quotient_mulg` yields the standard statement of the second isomorphism theorem.

In the statement of `third_isomorphism`, the term `H / K` is defined as `(coset_of K) @ H`. Since `K` is a group, `(coset_of K)` is equipped with a canonical structure of morphism. Since `H` is a group, its image by a morphism has itself a canonical structure of group, hence type inference is able to find a group structure for `H / K`, in the group domain `(coset K)`. Finally, this allows Coq to infer a group domain structure for `coset (H / K)`, the domain of the isomorphism, which is (hidden) implicit parameter of the `isog` predicate.

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: the Sylow theorems, the Frobenius lemma and the Cauchy-Frobenius lemma. The Sylow theorems are central in group theory. The Frobenius lemma gives a nice property of the elements of a group of a given order. Finally the Cauchy-Frobenius lemma, also called the 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 [6].

4.1 The Sylow Theorems

The first Sylow theorem asserts the existence of a subgroup H of K of cardinality p^n, for every prime p such that $\mathrm{card}(K) = p^n s$ and p does not divide s. For any two groups H and K of the same group domain, we define and prove :

Definition sylow K p H := H ⊂ K && card H ≡ $p^{\log_p(\mathrm{card\ K})}$.

Theorem sylow$_1$: ∀ K p, ∃ H, sylow K p H.

The first definition captures the property of H being a p-Sylow subgroup of K. The expression $\log_p$ (card K) computes the maximum value of i such that p^i divides the cardinality of K when p is prime. This theorem has already been formalised by Kammüller and Paulson [13], based a proof due to Wielandt [22]. 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 the unit acts trivially.

A complete account of our proof is given in [20]. The proof works by induction on n showing that there exists a subgroup of order p^i for all $0 < i \leq n$. The base case is Cauchy theorem, which asserts that a group K has an element of order p for each prime divisor of the cardinality of the group K. Our proof is simpler than the combinatorial argument used in [13], which hinged on properties of the binomial. We first build the set U such that $U = \{(k_1, \ldots, k_p) \mid k_i \in K \text{ and } \prod_{i=1}^{i=p} k_i = 1\}$. We have that $\mathrm{card}(U) = \mathrm{card}(K)^{p-1}$. We then define the action of the additive group $\mathbb{Z}/p\mathbb{Z}$ that acts on U as

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

Note that defining this action is straightforward since p-tuples are graphs of functions whose 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 $k \in \mathrm{K}$ and $k^p = 1$. A consequence of the orbit stabiliser theorem tells us that p divides the cardinality 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 p.

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

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

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

We have also formalised the second and third Sylow theorems. The second theorem states that any two 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 p-Sylow subgroups divides the cardinality of K and is equal to 1 modulo p. The third theorem is proved by acting by conjugation on the sets of all p-Sylow subgroups.

4.2 The 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. The 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 ϕ_gen: ∀ a, ϕ(orderg a) = card (generator (cyclic a)).

where ϕ 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 is 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 special 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 lemma. 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.

The action of an isometry p on 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 isometry.

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

5 Conclusion

To our knowledge, what is presented in this paper is already one of the most complete formalisations of finite group theory. We cover almost 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 [15]. Theorems like the ones presented in Section 4 are missing from the Mizar formalisation.

We have deliberately specialised our formalisation to *finite* group theory: finite groups are *not* obtained as a sub-class of generic groups. This design choice is consistent with the usual presentation of group theory in the literature. Imposing the inheritance of finite group theory from generic group theory would be somehow artificial since they share little results, all of them being trivial ones, hence it would not help formalising proofs but create pointless delicate subtyping issues.

Getting the definitions right is one of the most difficult aspects 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. It took much trial and error to arrive at 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. Decidable types and a substantial use of rewriting for our proofs give a 'classical' flavour to our development that is more familiar to what can be found in provers like Isabelle [17] or Hol [11] than what is usually done in Coq. Moreover, our novel use of canonical structures allows us to reconcile the convenience of set-theoretic statements with the expressiveness of dependent types, by harnessing the automation power of type inference. We think that this combination makes the Coq system a powerful environment for the formalisation of such algebraic theories.

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. Arthan, R.: Some group theory, available at `http://www.lemma-one.com/ProofPower/examples/wrk068.pdf`
2. Avigad, J., Donnelly, K., Gray, D., Raff, P.: A Formally Verified Proof of the Prime Number Theorem. ACM Transactions on Computational Logic (to appear)
3. Bailey, A.: Representing algebra in LEGO. Master's thesis, University of Edinburgh (1993)
4. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming 13(2), 261–293 (2003)
5. Bender, H., Glauberman, G.: Local analysis for the Odd Order Theorem. London Mathematical Society Lecture Note Series, vol. 188. Cambridge University Press, Cambridge (1994)
6. Constantine, G.M.: Group Theory, available at `http://www.pitt.edu/~gmc/algsyl.html`
7. Feit, W., Thompson, J.G.: Solvability of groups of odd order. Pacific Journal of Mathematics 13(3), 775–1029 (1963)
8. Geuvers, H., Wiedijk, F., Zwanenburg, J.: A Constructive Proof of the Fundamental Theorem of Algebra without Using the Rationals. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) TYPES 2000. LNCS, vol. 2277, pp. 96–111. Springer, Heidelberg (2002)
9. Gonthier, G.: A computer-checked proof of the four-colour theorem, available at `http://research.microsoft.com/~gonthier/4colproof.pdf`
10. Gonthier, G.: Notations of the four colour theorem proof, available at `http://research.microsoft.com/~gonthier/4colnotations.pdf`
11. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: a theorem proving environment for higher-order logic. Cambridge University Press, Cambridge (1993)
12. Gunter, E.: Doing Algebra in Simple Type Theory. Technical Report MS-CIS-89-38, University of Pennsylvania (1989)
13. Kammüller, F., Paulson, L.C.: A Formal Proof of Sylow's Theorem. Journal of Automating Reasoning 23(3-4), 235–264 (1999)
14. The Coq development team: The Coq proof assistant reference manual. LogiCal Project, Version 8.1 (2007)
15. The Mizar Home Page, `http://www.mizar.org/`
16. Paulin-Mohring, C.: Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation à diriger les recherches, Université Claude Bernard Lyon I (December 1996)
17. Paulson, L.C. (ed.): Isabelle. LNCS, vol. 828. Springer, Heidelberg (1994)
18. Peterfalvi, T.: Character Theory for the Odd Order Theorem. London Mathematical Society Lecture Note Series, vol. 272. Cambridge University Press, Cambridge (2000)
19. The Flyspeck Project, `http://www.math.pitt.edu/~thales/flyspeck/`
20. Rideau, L., Théry, L.: Formalising Sylow's theorems in Coq. Technical Report 0327, INRIA (2006)
21. Werner, B.: Une théorie des Constructions Inductives. PhD thesis, Paris 7 (1994)
22. Wielandt, H.: Ein beweis für die Existenz der Sylowgruppen. Archiv der Mathematik 10, 401–402 (1959)
23. Yu, Y.: Computer Proofs in Group Theory. J. Autom. Reasoning 6(3), 251–286 (1990)