A Machine-Checked Proof of the Odd Order Theorem

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al.

A Machine-Checked Proof of the Odd Order Theorem

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 and Laurent Théry

Microsoft Research - Inria Joint Centre

Abstract. This paper reports on a six-year collaborative effort that culminated in a complete formalization of a proof of the Feit-Thompson Odd Order Theorem in the Coq proof assistant. The formalized proof is constructive, and relies on nothing but the axioms and rules of the foundational framework implemented by Coq. To support the formalization, we developed a comprehensive set of reusable libraries of formalized mathematics, including results in finite group theory, linear algebra, Galois theory, and the theories of the real and complex algebraic numbers.

1 Introduction

The Odd Order Theorem asserts that every finite group of odd order is solvable. This was conjectured by Burnside in 1911 [34] and proved by Feit and Thompson in 1963 [14], with a proof that filled an entire issue of the Pacific Journal of Mathematics. The result is a milestone in the classification of finite simple groups, and was one of the longest proof to have appeared in the mathematical literature to that point. Subsequent work in the group theory community aimed at simplifying and clarifying the argument resulted in a more streamlined version of the proof, described in two volumes [6, 36]. The first of these, by H. Bender and G. Glauberman, deals with the "local analysis", a term coined by Thompson in his dissertation, which involves studying the structure of a group by focusing on certain subgroups. The second of these, by T. Peterfalvi, invokes character theory, a more "global" approach that involves studying a group in terms of the ways it can be represented as a group of matrices.

Both the size of this proof and the range of mathematics involved make formalization a formidable task. The last couple of decades have brought substantial advances in the use of interactive theorem provers, or "proof assistants," towards verifying substantial mathematical results [5, 16, 23]. The technology has also been used to verify the correctness of hardware and software components with respect to given specifications; significant successes in that area include the formal proof of correctness of a realistic compiler [32] and of a small operating system [29]. Formal methods have been especially useful when it comes to verifying the correctness of mathematical proofs that rely on computations that are

too long to be checked by hand. For example, Appel and Haken's proof of the four-color theorem [1] has been verified [16] in the Coq system [7], and Thomas Hales' *Flyspeck* project [23] is working towards verifying a proof of the Kepler conjecture, which Hales himself first established with contributions by Samuel Ferguson.

The formalization described in the present article, however, is of a different nature. The proof of the Odd Order Theorem does not rely on mechanical computation, and the arguments were meant to be read and understood in their entirety. What makes the formalization difficult — and interesting — is the combination of theories involved. Working with these theories formally required developing a methodology that makes it possible to switch, efficiently, between the various views a mathematical text can superimpose on the same mathematical object. Another important task has been to formalize common patterns of mathematical reasoning. When it comes to formal verification of software, interaction with a proof assistant is commonly based on case analysis and structural induction. In contrast, the proof of the Odd Order Theorem relies on a variety of argument patterns that require new kinds of support.

In Section 2 we outline the statement and proof of the Odd Order Theorem. Section 3 provides some examples of the design choices we adopted to represent mathematical concepts in the type theory underlying the Coq system. In Section 4 we review some examples of the techniques we used to represent efficiently different kinds of proof patterns encountered in this proof. In Section 5 we provide three examples of advanced mathematical theories whose formalization require a robust combination of several areas of formalized mathematics, before scaling to the main proof. Section 6 concludes the paper with comments and some quantitative facts about this work.

2 An overview of the Odd Order Theorem

2.1 Preliminaries

A *group* G consists of a set, usually also named G, together with an associative binary law $*$, usually denoted by juxtaposition, and an identity element 1, such that each element g of G has an inverse g^{-1}, satisfying $gg^{-1} = g^{-1}g = 1$. When there is no ambiguity, we identify an element g of a group with the corresponding singleton set $\{g\}$. In particular the trivial group $\{1\}$ is denoted by 1. The cardinality of G is called the *order* of the group. Examples of finite groups include the cyclic group $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n under addition, with identity 0; the set S_n of permutations of $\{0, \ldots, n-1\}$, under composition; and the set of isometries of a regular n-sided polygon. These examples have order n, $n!$, and $2n$, respectively. The cartesian product $G_1 \times G_2$ of two groups G_1 and G_2 is canonically a group with law $(a_1, a_2) * (b_1, b_2) := (a_1 b_1, a_2 b_2)$; the group $G_1 \times G_2$ is called the *direct product* of G_1 and G_2.

The law of an *abelian* group is commutative; in a non-abelian group G, we only have $ab = ba^b = ba[a, b]$, where $a^b := b^{-1}ab$ is the *b-conjugate* of a, and

$[a, b] := a^{-1}b^{-1}ab$ is the *commutator* of a and b. Product and conjugation extend to subsets A, B of G, with $AB := \{ab \mid a \in A,\ b \in B\}$ and $A^b := \{a^b \mid a \in A\}$. A subset A of G is *B-invariant* when $A^b = A$ for all b in B; in that case we have $AB = BA$.

One says that H is a *subgroup* of a group G, and writes $H < G$, when H is a subset of G containing 1 and closed under product and inverses — thus itself a group. For finite H, $H < G$ is equivalent to $1 \cup H^2 \subset H \subset G$. The set of subgroups of G is closed under intersection, conjugation, and commutative product (such as product with an invariant subgroup). If G is finite and $H < G$, then the order of H necessarily divides the order of G. It is not generally the case that for each divisor of the order of G there exists a subgroup of G of this order, but if G is a group of order n and p is a prime number dividing n with multiplicity k, then there exists a subgroup of G having order p^k, called a Sylow p-subgroup of G.

The notion of a *normal subgroup* is fundamental to group theory:

Definition 1 (Normal subgroup). *H is a* normal subgroup *of a group G, denoted $H \triangleleft G$, when H is a G-invariant subgroup of G.*

If $H \triangleleft G$, the set $\{Hg \mid g \in G\}$ of H-cosets is a group, as $(Hg_1)(Hg_2) = H(g_1g_2)$. This group, denoted G/H, is called the *quotient group* of G and H because it identifies elements of G that differ by an element of H. If G_1 and G_2 are groups, G_1 and G_2 are both normal in the group $G_1 \times G_2$.

Every finite abelian group is isomorphic to a direct product of cyclic groups $\mathbb{Z}/p_i^{k_i}\mathbb{Z}$, where the p_i are prime numbers. The far more complex structure of nonabelian groups can be apprehended using an analogue of the decomposition of a natural number by repeated division:

Definition 2 (Normal series, factors). *A* normal series *for a group G is a sequence $1 = G_0 \triangleleft G_1 \cdots \triangleleft G_n = G$, and the successive quotients $(G_{k+1}/G_k)_{0 \le k < n}$ are called the* factors *of the series.*

A group G is *simple* when its only proper normal subgroup is the trivial group 1, i.e., if its only proper normal series is $1 \triangleleft G$. A normal series whose factors are all simple groups is called a *composition series*. The Jordan-Hölder theorem states that the (simple) factors of a composition series play a role analogous to the prime factors of a number: two composition series of the same group have the same factors up to permutation and isomorphism. Unlike natural numbers, however, non-isomorphic groups may have composition series with isomorphic factors. The class of *solvable* groups is characterized by the elementary structure of their factors:

Definition 3 (Solvable group). *A group G is* solvable *if it has a normal series whose factors are all abelian.*

Subgroups and factors of solvable groups are solvable, so by the structure theorem for abelian groups, a finite group is solvable if and only if all the factors of its composition series are cyclic of prime order. We are now able to state the Odd Order Theorem.

2.2 The Odd Order Theorem

Theorem 1 (Odd Order theorem [14]). *Every finite group of odd order is solvable.*

It is striking that the theorem can be stated in such elementary terms, whereas its proof requires much more baggage. The file `stripped_Odd_Order`[1] of [39] provides a minimal, self-contained formulation of the Odd Order theorem, using only the bare Coq logic, and avoiding any use of extra-logical features such as notations, coercions or implicit arguments.

The proof of Theorem 1 proceeds by induction, showing that no minimal counterexample G exists. At the outset G is only known to be simple, nonabelian of odd order, but all proper subgroups of G should be solvable. The first half of the proof exploits these meager facts to derive a detailed description of the maximal proper subgroups of G, reducing the general structure of G to five cases. The second half of the proof uses character norm inequalities to rule out four of these, and extract some algebraic identities in a finite field from the last one. Galois theory is then used to refute these, completing the proof.

The study of the (solvable) subgroups of G exploits their decomposition into prime factors, reconstructing the structure of a maximal subgroup M from that of its p-factors for individual primes p. An A-invariant subgroup H of M has a normal series with A-invariant *elementary abelian* factors, that is, direct products of prime cycles. Identifying each one with a vector space over a finite field $\mathbf{F}_p$ makes it possible to analyze the action of A on H via the *representations* mapping A to a group of matrices over $\mathbf{F}_p$, and use linear algebra techniques such as eigenspace decomposition. Indeed, the proof starts by showing that 2×2 representations are abelian, then that no representation of A has a quadratic minimal polynomial (this replaces the use of the Hall-Higman theorem in [14]). This *p-stability* is combined with Glauberman's ZJ^* factorization to establish a Uniqueness theorem (Chapter II of [6]): any subgroup of *rank* 3 (containing an elementary abelian subgroup of order p^3) lies in a unique maximal subgroup of G.

Combining the Uniqueness theorem with results of Blackburn on odd groups of rank 2 yields that any maximal subgroup M of G is a *semidirect product* $M_\sigma E$ with $M_\sigma \triangleleft M$ and M_σ, E of coprime order. Furthermore, very few elements of M_σ and E commute — M is similar to a *Frobenius group*. Further analysis reveals that most M are of type I: M is very nearly a Frobenius group, with M_σ equal to the direct product M_F of the normal Sylow subgroups of M. However some M can be of type P, with $M = M_F U W_1$, where W_1 is cyclic, UW_1 is a Frobenius group, and all w_1 in W_1 commute precisely with the same cyclic group $W_2 < M_F$ (W_1 acts in a *prime manner* on M_F). Type P is subdivided into types V, II, III or IV, according to whether U is trivial, included in a different maximal group, abelian, or nonabelian, respectively. If any, there are exactly two type P groups up to conjugation, with W_1 and W_2 interchanged; at least one has type II, and over half of the elements of G lie in conjugates of $W = W_1 W_2$.

[1] http://coqfinitgroup.gforge.inria.fr/doc/stripped_odd_order_theorem.html

The second part of the proof [36] uses *characters*. The *character* of a complex representation $\rho : H \mapsto GL(n, \mathbb{C})$ is the function mapping each $h \in H$ to the trace of $\rho(h)$. In general, a character is *not* a group homomorphism, but it is a *class function*, constant on conjugacy classes of H. Convolution over H makes the set of class functions on a group H into a Hermitian space, for which the set irr H of *irreducible characters* of H forms an orthonormal basis. Characters of H have natural integer coordinates in irr H, hence integral norm.

Local analysis provides us both with a precise description of the characters of a maximal subgroup M, and an isometry mapping certain *virtual characters* of M (differences of characters) to virtual characters of G. This *Dade isometry* is only defined on functions that vanish on 1, so in order to extract usable information on G one needs *coherence* theorems extending it to a set of proper characters. The first, due to Sibley, covers Frobenius and type V maximal subgroups, and the second type II–IV subgroups.

For any $\chi \in$ irr G, coherence for a set (M_i) of non-conjugate maximal subgroups implies a numerical inequality bounding the sum of the (Hermitian) norms of the inverse images of the restrictions of χ to the support of the image of the Dade isometries for the M_i, and the (complex) norms of the values of χ elsewhere. For types III–V this bound yields a non-coherence theorem, which successively eliminates types V and IV; this implies that type I groups are actually Frobenius, and then the coherence bound forces type P groups to exist.

More inequalities then force the M_F, U, and W_1 subgroups of the type P groups to be isomorphic to the additive, unitary multiplicative, and Galois groups of a finite field $\mathbf{F}_{p^q}$ of order p^q, then rule out type III, and imply that U is W_2^y-invariant for some $y \in H_F$, where H is the *other* type II group such that $W_1 < H_F$. Intricate calculations show that this implies that if $a \in \mathbf{F}_{p^q}$ and $2 - a$ both have Galois norm 1, then so does $\tau(a) := 2 - 1/a$, and hence $\tau(a) \dots \tau^k(a) = (1 - 1/a)k + 1$; for $a \neq 1$ the Galois norm of $(1 - 1/a)x + 1$ yields a polynomial of degree q which has $0, ..., p - 1$ as roots, whence $q \leq p$ and hence $q = p$ by symmetry, so the orders of M_F and $E > W_1$ are not coprime, a contradiction.

2.3 Mathematical sources

Our formalization follows the two books describing the revised proof [6, 36], with a small number of exceptions, for which we formalized the original arguments [14]. We followed Huppert's proof [27] of the Wielandt fixed point order formula. The elementary finite group theory part follows standard references [31]. For more advanced material we have used Aschbacher's and Gorenstein's books [3, 22] and some sources adapted to Galois theory and commutative algebra [30, 33, 37]. The formalization of character theory is based on Isaacs [28].

3 Mathematical structures and interfaces in Coq

3.1 The Calculus of Inductive Constructions and the Coq system

Most mathematical papers and textbooks do not explicitly specify a formal axiomatic foundation, but can generally be viewed as relying on set theory and

classical logic. Many interactive theorem provers, however, use alternative formal systems based on some form of type theory. Just as in the domain of programming languages, types help classify expressions passed to the checker, and hence facilitate the verification of claims in which they appear. The Coq proof assistant [7] is based on a logical foundation known as the *Calculus of Inductive Constructions*, or *CIC* [11, 12], a powerful and expressive version of constructive dependent type theory.

The advantage to using dependent type theory is that types can express complex specifications. For example, in our formalization, if G is an object of type `finGroupType`, then G is a record type which packages the type representing the elements of G, the binary group operation, the identity, and the inverse, as well as proof that these items satisfy the group axioms. In addition, Coq's type inference algorithm can eliminate the need to provide information that can be reconstructed from type constraints, just as implicit information is reconstructed by an experienced reader of a page of mathematics. For example, if g and h are elements of the carrier type of G an object of type `finGroupType`, then when presented with the expression $g * h$, Coq can infer that $*$ denotes the binary operation of G, as well as the fact that that operation is associative, and so on. Thus, type inference can be used to discover not only types, but also data and useful facts [19, 4]. Working with such an elaborate type system in a proof assistant can be delicate, however, and issues like the decidability of type checking impose severe restrictions on the nature of the dependent types one can work with in practice.

The status of computation in Coq's formalism also plays a central role in the present formalization. Every term or type has a computational interpretation, and the ability to unfold definitions and normalize expressions is built in to the underlying logic. Type inference and type checking can take advantage of this computational behavior, as can proof checking, which is just an instance of type checking. The price to pay for this powerful feature is that Coq's logic is constructive. In Coq many classical principles, such as the law of the excluded middle, the existence of choice functions, and extensionality are not available at the level of the logic. These principles can be recovered when they are provably valid, in the constructive sense, for specific objects like finite domains, or they can be postulated as axioms if needed. The present formalization, however, does not rely on any such axiom. Although it was not the primary motivation for this work, we eventually managed to obtain a completely constructive version of the proof and of the theories it requires.

In short, the success of such a large-scale formalization demands a careful choice of representations that are left implicit in the paper description. Taking advantage of Coq's type mechanisms and computational behavior allows us to organize the code in successive layers and interfaces. The lower-level libraries implement constructions of basic objects, constrained by the specifics of the constructive framework. Presented with these interfaces, the users of the higher-level libraries can then ignore these constructions, and they should hopefully be able to describe the proofs to be checked by Coq with the same level of comfort

as when writing a detailed page in LaTeX. The goal of this section is to describe some of the design choices that were made in that respect.

3.2 Principles of boolean reflection

The equality relation $x = y$ on a given type is generally not decidable, which is to say, the alternative $x = y \vee x \neq y$ is not generally provable. In some contexts, one can prove that equality is equivalent to a boolean relation $x \equiv y$, for which the law of the excluded middle holds. In our formalization, an `eqType` is a type paired with such a relation. Working with an `eqType` thus provides a measure of classical reasoning.

Using boolean values and operations to reason about propositions is called *boolean reflection*. This makes it possible, in a sense, to "calculate" with propositions, for example, by rewriting with boolean identities. More generally, to take advantage of propositions that can be represented as boolean values, the libraries provides infrastructure theorems to link logical connectives on (boolean) propositions with the corresponding boolean connectives. The SSReflect tactic language also provides support to facilitate going back and forth between different but equivalent descriptions of the same notion, like between boolean predicates and their logical equivalents. An explicit coercion is used throughout the formalization, which inserts automatically and silently an injection from a boolean value `b` to the formula `(b = true)` in the type Prop of propositions. This strategy is central to our methodology, and explains the name SSReflect, which is short for *small scale reflection* [18, 20].

Decidable equality plays another important role in facilitating the use of subtypes. If A is a type, a subset of A can be represented by a predicate B on A, that is, a map B from A to the type Prop. An element of this subset can be represented by an element a of A, and a "proof" p that B holds for a. The dependent pair $\langle a, p\rangle$ is an element of the *dependent sum*, or *Sigma type*, $\Sigma_{x:A} Bx$. As B takes values in Prop, $\Sigma_{x:A} Bx$ is also called a *subtype* of A, for the reasons just described.

The problem is that the equality on a subtype is not as simple as one would like. Two elements $\langle a_1, p_1\rangle$ and $\langle a_2, p_2\rangle$ are equal if and only if $a_1 = a_2$ but also, now considering both p_1 and p_2 as proofs that a_1 satisfies B, p_1 and p_2 are the *same* proof. However, a theorem due to Hedberg [25], formalized in our library as `eq_irrelevance`[2], implies that if B is a boolean-valued rather than Prop-valued predicate, then any two proofs p_1 and p_2 that a_1 satisfies B are equal. Thus, in this case, two elements $\langle a_1, p_1\rangle$ and $\langle a_2, p_2\rangle$ of the subtype are equal if and only if $a_1 = a_2$. Our libraries provide support [18] for the manipulations of these boolean subtypes, which are used pervasively in the formalization.

3.3 Finite group theory

Given the substantial amount of group theory that needed to be formalized, we relied on two important observations to optimize the data structures used to

[2] http://coqfinitgroup.gforge.inria.fr/doc/eqtype.html

represent finite groups. First, in many proofs, one can take most or all of the groups involved to be subgroups of a larger ambient group, sharing the same group operation and identity. Moreover, local notions like the normalizer of H inside of G, denoted $\mathrm{N}_G(H)$, are often used "globally," as in $\mathrm{N}(H)$, a practice which implicitly assumes that the normalizer is to be computed within an ambient container group. Second, and more importantly, many theorems of group theory are equally effective when stated in less generality, in terms of subgroups of such an ambient group. For example, given a theorem having to do with two unrelated groups, it does not hurt to assume that the two groups are subgroups of a larger group; this can always be made to hold by viewing them as subgroups of their direct product.

In our formalization, we represent such ambient groups as `finGroupType`s, and then represent the groups of interest as subsets of that type that contain the identity and are closed under the group operation. This is much simpler than having to maintain a plurality of types and morphisms between them. We form a new `finGroupType` only when strictly necessary, for example when forming a quotient group, which requires a new group operation [19].

A delicate point we had to cope with is that many constructions are partial. For example, the quotient group G/N can be formed only if $N \triangleleft G$ (see Section 2.1), and the direct product $G \times H$ of two subgroups of an ambient group can be formed only if they commute and $G \cap H = 1$. In addition, given our encoding of groups as subsets of a container group type, morphisms are unlikely to be defined on the whole group type, but rather on a specific subset.

Such constructions are ubiquitous. Forming subtypes as described in Section 3.2 in each case would be unwieldy and would make the application of lemmas mentioning partial constructions particularly cumbersome. The general approach has been to make each of these constructions total, either by modifying the input when it is invalid, or returning a default value. For example, the direct product construction returns the empty set (which is not a group) if the input groups have a nontrivial intersection; applying a morphism to a set automatically shrinks the input set by intersecting it with the domain of the morphism. The downside of this approach is that lemmas involving partial constructions often, but not always, depend on side conditions that need to be checked when the lemma is applied.

3.4 Dependent records as first class interfaces

Records are just a generalization of the dependent pair construction described in Section 3.2, and, in the same way, can be used to package together types, data, operations and properties. They can thus play the role of abstract "interfaces." Such interfaces are very natural in abstract algebra, but are also useful in developing a theory of iterated operations [8], a theory of morphisms, a theory of algebraic structures [15] and so on. For an extensive list of interfaces used in the SSReflect library, the reader can refer to Section 11.3 of the SSReflect documentation [20].

In addition to the hierarchy of algebraic structures, we also provide a hierarchy for numeric fields [9], which are fields equipped with a boolean order relation, possibly partial. The purpose of this small hierarchy is to capture the operations and the theory of the complex number field and its subfields (cf Section 5.2).

Here we simply provide an example to illustrate how, using record types and setting up type inference carefully, one can obtain a hierarchy of interfaces that provides multiple inheritance, notation overloading, and (as we will see in the next section) a form of proof search. Consider the expression `(x + x * x == 0)`, where `x` is taken to be of type `int`. The symbols `*`, `+` and `==` are overloaded, and correspond to the multiplication of a ring, the addition operation in an additive group, and a decidable comparison operation. Type inference has to check that operations are applied to arguments of the right type; for example `x` of type `int` is used as an argument of `*`, hence the type `int` is *unified* with the carrier of an unspecified ring structure. Unification can be programmed, thanks to Coq's *canonical structures* mechanism, to solve such a unification problem by fixing the unknown structure to be the canonical ring structure on the integers. Given that `int` has been proved to be an instance of all the structures involved, unification always succeeds and type inference can make sense of the input expression, binding the overloaded symbols to the respective integer operations.

3.5 Searching proofs by programming type inference

Very often, the verification of small, uninteresting details is left implicit in an ordinary mathematical text, and it is assumed that a competent reader can fill these in. Canonical structures can be programmed to play a similar role. In particular, structures can package data with proofs, as in Section 3.2, in which case searching for a particular structure that contains a certain value can amount to looking for a proof that some property holds for that value.

A slight difficulty is that canonical structures are designed to guide unification, which is used by type inference to process *types* (like `int`), while here we need to process *values* (like the intersection of two sets). The crucial observation is that Coq's logic features dependent types, which means that values can be injected into types, even artificially, to make them available to unification, and hence to canonical structure resolution. We call the mechanism for doing this a *phantom type*. The use is similar to the use of phantom types in functional programming [26], where one enriches a type with a dummy (phantom) annotation to trick the type system into enforcing additional invariants.

Manifestations of this automatic proof search machinery are ubiquitous. For example, we can prove `(1 \in f @* (G :&: H))` by applying the `group1` lemma, which states that a set contains the unit element `1` if it happens to be a group. The canonical structure mechanism infers this group structure automatically for `f @* (G :&: H)`: if `G` and `H` have a group structure, then so does their intersection, as well as the image of that intersection under a group morphism `f`.

An advanced use of canonical structures is found in the definition of the `mxdirect` predicate, which states that its argument is a finite sum $\sum_{i=1}^{n} E_i$ of

vector spaces $(E_i)_{i\in[1\ldots n]}$ that is moreover direct. What makes the `mxdirect` predicate unusual is that it is computed from the *syntax* of its argument, by comparing the rank $r(\sum_{i=1}^{n} E_i)$ of the vector space defined by the whole expression with the sum $\sum_{i=1}^{n} r(E_i)$ of the ranks of its components. If the two numbers are equal, the sum is direct. The canonical structure mechanism is programmed to recognize the syntax of an arbitrary sum of vector spaces, to collect the single spaces, to sum up their ranks. The `mxdirect` predicate is then defined as the comparison of the result with the rank of the whole initial expression [17].

The canonical structures inference mechanism essentially provides a Prolog like resolution engine that is used throughout the libraries to write statements that are more readable and easier to use. Programming this resolution engine can be quite tricky, and a more technical explanation would go beyond the scope of this paper [21].

4 Mathematical proofs in Coq

4.1 Symmetries

One commonly invokes a symmetry argument in a mathematical proof by asserting that "without loss of generality" some extra assumption holds. For example, to prove a statement $P(x, y)$ in which x and y play symmetric roles, adding the assumption $x \leq y$ does not change the resulting theorem. Whereas an ordinary mathematical proof will leave it to the reader to infer the tacit argument, when doing formal proofs, it is useful to have support to fill in the details [24].

The SSReflect proof language provides a simple but effective tool in that regard: the `wlog` tactic [20]. This tactic performs a logical cut with a formula constructed from the *names* of the context items involved in the symmetry argument and the statement of the extra property the symmetry will exploit, both provided by the user. The logical cut generated by the proof shell involves the selected piece of context and the ongoing, usually very large, goal. For instance, when attempting to prove the statement `(P a b)`, the command `wlog H: a b / a <= b`, generates a first subgoal requiring a proof that `(H : forall x y, x <= y -> P x y)` holds, and another one to prove `(P a b)` under the assumption of `H`, which boils down to two applications of `H` if the statement `(P a b)` is actually symmetric in `a` and `b`. This simple tool has been instrumental at several places of the formalization, especially in large proofs using character theory [36].

4.2 Cycles of inequalities

A standard pattern of reasoning seems to conclude out of blue that some assertion holds, from a proof that a chain of nonstrict *inequalities* in which the first and last terms are the same. The implicit content is a three-step proof: the circularity of the chain forces each inequality to be an equality; for each inequality, the equality case is characterized by a certain condition; hence the conjunction of these conditions holds and the desired statement follows from these. Typical

examples of such inequalities come from the properties of convex functions, e.g., the inequality between the arithmetic and geometric means is related to the strict convexity of the exponential function. The equality conditions can, however, be more elaborate; for example, the rank of a sum of finite dimensional vector spaces is smaller than the sum of the ranks of the summed vector spaces and equality holds if and only if the sum is direct.

In order to formalize this kind of proof efficiently, the SSReflect library uses notation to pair an inequality with the condition under which equality holds. For example, consider the following lemma:

```
Lemma nat_Cauchy m n : 2 * (m * n)<= m ^ 2 + n ^ 2 ?= iff (m == n)
```

This hides the conjunction of the following statements:

```
2 * (m * n) <= m ^ 2 + n ^ 2
(2 * (m * n) == m ^ 2 + n ^ 2) = (m == n)
```

The second statement is an equality reflecting the equivalence of the two boolean statements. Because the `rewrite` tactic can take multi-rules as arguments [20], rewriting with `nat_Cauchy` can affect several kinds of comparisons. The library provides support for using these statements, including the transitivity lemma collecting the equality conditions that is instrumental in capturing the pattern of reasoning described above. This technique has been a key ingredient in the formalization of advanced results[3] using character theory [36].

4.3 Proof search by large-scale reflection

Most of the proofs we worked from were not amenable to automation. A notable exception is found in Section 3 of the second volume of the proof [36], which deals with an indexed family of *virtual characters* (β_{ij}). These are defined to be integer linear combinations $\beta_{ij} = \sum_k z_k \chi_k$ of irreducible characters, where the collection of irreducible characters (χ_k) form a family of class functions that is orthonormal for the inner product $\langle \cdot, \cdot \rangle$. The array of virtual characters in question satisfies certain combinatorial constraints:

- For each i, $\langle \beta_{ij}, \beta_{ij} \rangle = 3$.
- For any two distinct elements on the same row or the same column ($i = i'$ or $j = j'$ but not both), $\langle \beta_{ij}, \beta_{i'j'} \rangle = 1$.
- Any two elements on distinct rows and columns ($i \neq i'$ and $j \neq j'$) are orthogonal, that is, $\langle \beta_{ij}, \beta_{i'j'} \rangle = 0$.

These conditions impose tight constraints on the β_{ij}'s.

A two-page combinatorial argument [36] shows that if there are at least four rows and two columns, then elements of the same column have a common irreducible character, and the respective coefficients are equal. We initially formalized this argument by hand. Later, Pascal Fontaine (one of the developers of the SMT solver veriT), provided us with an encoding[4] that made it possible

[3] See, for instance, http://coqfinitgroup.gforge.inria.fr/doc/PFsection9.html.

[4] The SmtLib file is available at http://coqfinitgroup.gforge.inria.fr/smt/th3_5.smt.

to automate the proof using a trusted connection between Coq and an SMT solver [2]. In order to have an automated version of the proof within the Coq system, we ultimately encoded the proof search directly, taking advantage of the symmetry of the problem. This was done using large-scale reflection, supported by notation-based reification. This automated version is shorter than our initial version, compiles twice as fast, and is intellectually more satisfying, as it eliminates unnecessary steps from the original proof [36].

4.4 Classical reasoning

The instances of the mathematical structures of our hierarchy (see Section 3.4) are required to have boolean operators for the comparison and, possibly, the ordering of their inhabitants. We provide instances for all these structures, and all the instances needed for this formalization are in fact either finite types or countable types. We therefore benefit from other classical properties otherwise not available in the constructive logic of Coq. Indeed, countable types satisfy the functional choice axiom for boolean predicates (Markov's principle); functions on finite types can be represented by their graphs, which are extensional: the graphs of any two functions that are pointwise equal are in fact equal [18].

Boolean reflection extends to any first-order theory that has a decision procedure. In particular, the algebraic hierarchy mentioned in Section 3.4 has an interface for fields with a decidable first-order theory. The specification of this property uses a deep embedding of first-order formulas together with a boolean satisfiability predicate. Finite fields are of course instances of this interface, as are algebraically closed fields, which enjoy quantifier elimination [38, 10]. Decidable fields are used in the formalization of representation theory, both in dealing with modular representations, which are based on finite fields, and complex representations, which are based on algebraic complex numbers.

In other cases first-order decidability fails, notably for the rationals and for number fields. As a result, we elected not to rely on this interface for some basic results in the theory of group modules that cannot be proved constructively. Instead, we proved their double negation, expressed using the `classically` monadic operator [17, 35]:

```
Definition classically (P : Prop) : Prop :=
 forall b : bool, (P -> b) -> b.
```

Note the implicit use in this statement of the coercion mentioned in Section 3.2. The statement `(classically P)` is logically equivalent to `(~~ P)`, but this formulation is more useful in practice, because when using a hypothesis of the form `(classically P)` in the proof of a statement expressed as a boolean (on which excluded middle holds), one can constructively assume that `P` itself holds.

The `classically` operator is used only in the file formalizing the theory of group modules for representations. Note that although we use the `classically` operator to weaken the statement of some theorems we formalized, we did not need to alter the statement of Odd Order Theorem to describe its proof completely within the calculus of inductive constructions.

5 Mathematical theories

5.1 Representations and characters

Our treatment of linear algebra is organized in two levels. On the abstract level, a hierarchy of structures (see Section 3.4) provides interfaces, notations and shared theories for vectors, F-algebras and their morphisms. On the concrete level, these structures are instantiated by particular models, centered on matrices [17]. The central ingredient to this latter formalization is an extended Gaussian elimination procedure similar to LUP decomposition. This formalization of matrix algebra itself contains proofs of nontrivial results, covering determinants, Laplace expansion for cofactors, and properties of direct sums.

The choices we have made are validated by the successful formalization of representation theory, which depends on both finite group theory and linear algebra. Thanks to the underlying formalization of linear algebra in terms of concrete matrices, it is fairly easy to define the representations of a given group G in terms of square matrices with coefficients in a given field F, as well as other important notions, like the enveloping algebra of a representation, or a group module. Part of the theory of group modules requires the extra assumption that F has a decidable first-order theory. The library includes formal proofs of the fundamental results of representation theory, including Schur's lemma, Maschke's theorem, the Jacobson density theorem, the Jordan-Hölder theorem, Clifford's theorem, the Wedderburn structure theorem for semisimple rings, etc.

The next step is the finite group character theory, the main prerequisite for the second part of the proof [36]. Characters are defined as class functions with complex values, equipped with their standard convolution product. We first define the tuple of class functions on a given group G that are irreducible characters of G; then characters are class functions that are linear combinations of irreducible characters with natural number coordinates. Finally, we define virtual characters: class functions that are integer linear combinations of a given list of class functions. All these definitions are constructive, thanks to the finiteness of the group, the decidability of the first order theory of the coefficient field (Section 4.4) (here the complex algebraic numbers) and the Smith normal form for integer matrices. The formalization includes results like the theory of inertia groups, Burnside's $p^a q^b$ theorem, and Burnside's vanishing theorem [28].

5.2 Complex algebraic numbers

Our formalization of the character theory used in the second volume of the proof [36] is parametrized by a decidable field of complex numbers. In order to provide a concrete instance of this interface, we formalized a construction of the algebraic numbers. Standard presentations of character theory use arbitrary complex numbers, but as characters can only take algebraic values, the restriction is innocuous.

The algebraics can be described as an algebraic closure of the rationals equipped with an involutive conjugation automorphism $z \mapsto \bar{z}$. The latter yields both a norm ($|z| = t\bar{t}$ for some $t^2 = z$) and a partial order ($x \leq y$ if $|y-x| = y-x$)

whose restriction to the real (conjugation-invariant) algebraics is total; that is, an implementation of our "numeric field" interface (Section 3.4).

We first obtained the algebraics as the complex extension $R[i]$ of the real closed field of real algebraic numbers R which we had constructed explicitly [9]; we adapted an elementary proof of the Fundamental Theorem of Algebra (FTA) based on matrix algebra [13] to show that $R[i]$ is algebraically closed.

We then refactored that construction to eliminate the use of the real algebraics. We construct the algebraics directly as a countable algebraic closure, then construct conjugation by selecting a maximal real subfield. Because we are within a closure we can use Galois theory and adapt the usual proof of the FTA to show that conjugation is total.

5.3 Galois theory

Galois theory establishes a link between field extensions and groups of automorphisms. A field extension is built by extending a base field with roots of polynomials that are irreducible on this base field. The vector space structure of such an extension plays an important role. The remarks and methods described in Section 3.3 apply in this situation: instead of assigning a type to each new extension, field extensions are formalized as intermediate fields between a fixed base field F and a fixed ambient splitting field extension L. A splitting field extension of F is a field extension of F generated by an explicit finite list of *all* the roots of a given polynomial.

All the constructions of this formalized Galois theory hence apply to extensions of a field F that are subfields of a field L. If K and E are intermediate extensions between F and L, the Galois group type (see Section 3.3) of E is the type of automorphisms of E. Then the Galois group $\mathrm{Gal}(E/K)$ of a field extension E/K is the set of automorphisms of the Galois group type that fix K. Partiality issues are dealt with in a manner similar to their treatment in finite group theory (see Section 3.3): the definitions take as arguments subspaces of the ambient field, but the theory is available for those vector spaces that are fields, a fact that can generally be inferred via a canonical structure. For example, $\mathrm{Gal}(E/K)$ is a set when E and K are vector spaces, but is equipped with a group structure as soon as E is a field.

It is interesting to note that standard Galois theory is usually carried out on normal extensions rather than on splitting fields. While the two notions are constructively equivalent, splitting fields are much easier to construct in practice.

6 Conclusion

The success of the present formalization relies on a heavy use of the inductive types [12] provided by Coq and on various flavors of reflection techniques. A crucial ingredient was the transfer of the methodology of "generic programming" to formal proofs, using the type inference mechanisms of the Coq system.

Our development includes more than 150,000 lines of proof scripts, including roughly 4,000 definitions and 13,000 theorems. The roughly 250 pages of

mathematics in our two main sources [6, 36] translate to about 40,000 lines of formal proof, which amounts to 4-5 lines of SSReflect code per line of informal text. During the formalization, we had to correct or rephrase a few arguments in the texts we were following, but the most time-consuming part of the project involved getting the base and intermediate libraries right. This required systematic consolidation phases performed after the production of new material. The corpus of mathematical theories preliminary to the actual proof of the Odd Order theorem represents the main reusable part of this work, and contributes to almost 80 percent of the total length. Of course, the success of such a large formalization, involving several people at different locations, required a very strict discipline, with uniform naming conventions, synchronization of parallel developments, refactoring, and benchmarking for synchronization with Coq.

As we have tried to make clear in this paper, when it comes to formalizing this amount of mathematics, there is no silver bullet. But the combined success of the many techniques we have developed shows that we are now ready for theorem proving in the large. The outcome is not only a proof of the Odd Order Theorem, but also, more importantly, a substantial library of mathematical components, and a tried and tested methodology that will support future formalization efforts.

Acknowledgments. The authors would like to thank the Coq team for their continuous development, improvement and maintenance of the Coq proof assistant.

References

[1] K. Appel and W. Haken. Every map is four colourable. *Bulletin of the American Mathematial Society*, 82:711–712, 1976.
[2] M. Armand et al. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, volume 7086 of *LNCS*, pages 135–150, 2011.
[3] M. Aschbacher. *Finite Group Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2000.
[4] J. Avigad. Type inference in mathematics. *Bulletin of the European Association for Theoretical Computer Science. EATCS*, (106):78–98, 2012.
[5] J. Avigad and J. Harrison. Formally verified mathematics. To appear in the *Communications of the ACM*.
[6] H. Bender and G. Glauberman. *Local analysis for the Odd Order Theorem*. Number 188 in London Mathematical Society, LNS. Cambridge University Press, 1994.
[7] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: The calculus of inductive constructions*. Springer-Verlag, Berlin, 2004.
[8] Y. Bertot, G. Gonthier, S. Ould Biha, and I. Pasca. Canonical big operators. In *TPHOLs*, volume 5170 of *LNCS*, pages 86–101, 2008.
[9] C. Cohen. Construction of real algebraic numbers in coq. In *ITP*, volume 7406 of *LNCS*, pages 67–82, 2012.
[10] C. Cohen and A. Mahboubi. A formal quantifier elimination for algebraically closed fields. In *CICM*, volume 6167 of *LNCS*, pages 189–203, June 2010.
[11] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.
[12] T. Coquand and C. Paulin-Mohring. Inductively defined types. In *Colog'88*, volume 417 of *LNCS*. Springer-Verlag, 1990.

[13] H. Derksen. The fundamental theorem of algebra and linear algebra. *American Mathematical Monthly*, 100(7):620–623, 2003.
[14] W. Feit and J. G. Thompson. Solvability of groups of odd order. *Pacific Journal of Mathematics*, 13(3):775–1029, 1963.
[15] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *TPHOLs*, volume 5674 of *LNCS*, pages 327–342. Springer, 2009.
[16] G. Gonthier. Formal proof—the Four Color Theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.
[17] G. Gonthier. Point-free, set-free concrete linear algebra. In *ITP*, volume 6898 of *LNCS*, pages 103–118. Springer, 2011.
[18] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
[19] G. Gonthier, A. Mahboubi, L. Rideau, E. Tassi, and L. Théry. A Modular Formalisation of Finite Group Theory. In *TPHOLs*, pages 86–101, 2007.
[20] G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2012.
[21] G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, pages 163–175, 2011.
[22] D. Gorenstein. *Finite Groups*. AMS Chelsea Publishing Series. 2007.
[23] T. Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.
[24] J. Harrison. Without Loss of Generality. In *TPHOLs*, volume 5674 of *LNCS*, pages 43–59, 2009.
[25] M. Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, July 1998.
[26] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming, Cornerstones of Computing*, pages 245–262, 2003.
[27] B. Huppert and N. Blackburn. *Finite Groups II*. Grundlehren Der Mathematischen Wissenschaften. Springer London, Limited, 1982.
[28] I. Isaacs. *Character Theory of Finite Groups*. AMS Chelsea Pub. Series. 1976.
[29] G. Klein et al. sel4: formal verification of an os kernel. In *SOPS ACM SIGOPS*, pages 207–220, 2009.
[30] K. Konrad. Separability II. http://www.math.uconn.edu/~kconrad/blurbs/galoistheory/separable2.pdf.
[31] H. Kurzweil and B. Stellmacher. *The Theory of Finite Groups: An Introduction*. Universitext Series. Springer, 2010.
[32] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM Press, 2006.
[33] R. Mines, F. Richman, and W. Ruitenburg. *A course in constructive algebra*. Universitext (1979). Springer-Verlag, 1988.
[34] P. M. Neumann, A. J. S. Mann, and J. C. Tompson. *The collected papers of William Burnside, volume I*. Oxford University Press, 2004.
[35] R. O'Connor. Classical mathematics for a constructive world. *Mathematical Structures in Computer Science*, 21:861–882, 2011.
[36] T. Peterfalvi. *Character Theory for the Odd Order Theorem*. Number 272 in London Mathematical Society, LNS. Cambridge University Press, 2000.
[37] J. Rotman. *Galois Theory*. Universitext (1979). Springer, 1998.
[38] A. Tarski. A Decision Method for Elementary Algebra and Geometry, 1948. 2nd edition Berkeley, CA: University of California Press, 1951.
[39] The Mathematical Component Team. A Formalization of the Odd Order Theorem using the Coq proof assistant, September 2012. http://www.msr-inria.inria.fr/Projects/math-components/feit-thompson.