Exploring the capabilities of the Lean interactive theorem prover

Adrián Doña Mateo

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2023

Abstract

Lean is one of the latest additions to the family of theorem provers. We present an introduction to Lean and how it can be used to formalise mathematics. In the process, we formalise the solutions to four International Mathematical Olympiad problems and produce a careful development of the theory of composition series of groups and the Jordan–Hölder theorem for finite groups. This report provides both a guide to Lean's main features and an extensive discussion of its capabilities, through examples taken from these formalisation projects. We reflect on what features make Lean a useful tool for expressing mathematical content, and identify what areas need to be improved in order to make the formalisation process more natural.

Acknowledgements

I would like to thank my supervisor, Dr Paul Jackson, for introducing me to the world of formalised mathematics, and for his invaluable guidance throughout the realisation of this project.

For making it possible to learn to use Lean in a few months by answering all of my questions, I would like to thank Ramon Fernández Mir and the welcoming community of Lean users on Zulip. This project would have not been possible without them.

Lastly, I would like to extend my gratitude to my family and friends for their continuous support throughout this year.

Table of Contents

1 Introduction **1**
1.1 Computers and theorem proving . . . 1
1.2 The Lean theorem prover . . . 2
1.3 Objectives and contributions . . . 3

2 Introduction to Lean **4**
2.1 Lean's type theory . . . 4
2.2 The tactic system . . . 6
2.3 Type classes . . . 6

3 Formalising solutions to IMO problems **8**
3.1 Sequences of integers: 2014 question 1 . . . 9
3.2 Recurrence relations: 2017 question 1 . . . 12
3.3 Functional equations: 2017 question 2 . . . 13
3.4 Multivariate polynomials: 2017 question 6 . . . 14

4 Composition series and the Jordan–Hölder theorem **17**
4.1 Out of a textbook: a first definition . . . 18
4.2 Change in perspective: a second definition . . . 20
4.3 Developing the theory . . . 23
4.3.1 Existence of a composition series . . . 26
4.3.2 The second isomorphism theorem . . . 30
4.3.3 Uniqueness: the Jordan–Hölder theorem . . . 33
4.4 The fundamental theorem of arithmetic . . . 36

5 Conclusion **39**
5.1 Further work . . . 40

References **41**

Chapter 1

Introduction

The theory of mathematics has historically been confined to textbooks and lectures, written in a formal, albeit natural language. Ever since the advent of computers, mathematicians and computer scientists alike have pioneered ways of recording mathematical knowledge and reasoning in a way that machines can understand. In this project, we will explore one of the latest offspring of this vision: the Lean theorem prover.

1.1 Computers and theorem proving

The idea of using computers to formalise and prove mathematical statements has a long history. As early as the 1950s, mathematicians and computer scientists toyed with the idea of designing computer programs that could prove theorems automatically – see [1] and [2]. H. Wang [3] created three programs for an IBM 704 computer, the first of which could prove all theorems in the first five chapters of Whitehead and Russell's *Principia Mathematica* [10] in only thirty-seven minutes. Feats like these paved the way for many developments in the area of *automated theorem proving*. However, these programs, which would try to derive a given conclusion from a set of axioms using fixed logical inference rules, were mostly inefficient and relied on heuristics tuned by the developers [16].

It was only after time-sharing and personal computers became available in the late 1960s that *interactive theorem proving* (as opposed to automated) emerged. In this new paradigm, the human is responsible for the mathematical insight of the proof while the computer is tasked with filling in the formal gaps and checking its validity. This entailed the creation of a language through which the user could communicate her proof to the machine. Early examples of this are the SAM (Semi-Automated Mathematics) provers. These culminated with SAM V [4], which was used to solve an open problem in lattice theory. Proof assistants, as interactive theorem provers are also known, could not only be used for mathematical research, but also to formally express the specifications of complex engineering systems and prove their correctness.

An important family of proof assistants takes an approach based on the *Curry-Howard isomorphism* [5] to build their proofs. This describes an analogy between a system of logic and a programming language where to each proposition corresponds a type. In this sense,

a proof of a given proposition is identified with an object of the corresponding type. The logical connective of implication becomes the function type: if α and β are propositions-types, a proof of $\alpha \Rightarrow \beta$ is a function of type $\alpha \to \beta$, which, given a proof of α, produces a proof of β. Similarly, conjunction and disjunction become the product and sum types. This approach, that was pioneered by de Bruijn in his *Automath* proof assistant [6], was the stepping stone towards a big family of theorem provers that use type theory as their logical foundation. This includes NuPRL, Coq and Lean. Other logical foundations have also been successfully used, such as higher-order logic (HOL, HOL Light, Isabelle/HOL) or first-order set theory (Mizar, Isabelle/ZF).

The first applications of the isomorphism to theorem provers created the need for more powerful systems of types that led to the birth of *type theory*. Martin-Löf introduced a predicative type theory [11] that included inductive types (we will meet these inside Lean). This was later combined by Coquand with the polymorphic lambda calculus of Girard and Reynolds to create the *Calculus of Constructions* [7], later extended to the *Calculus of Inductive Constructions* [9]. The latter is the basis of the type theory of Coq and Lean.

Since they were released, users of most popular proofs assistants have incrementally built extensive libraries of formalised mathematics. Perhaps the largest dedicated exclusively to mathematics is Mizar's, with over 62,000 theorems and close to 13,000 definitions. Isabelle's Archive of Formal Proofs contains over 165,000 theorems, in areas including computer science, logic and mathematics. Coq's Mathematical Components library includes over 13,000 theorems, and it serves as the basis for numerous other projects. This library originated from Georges Gonthier's formalisation of the four-colour theorem [15], and was later used in the formalisation of the odd order theorem [19], which are two of the greatest achievements of formalised mathematics. Lean's mathematical library is rapidly growing, and now has over 52,000 theorems and 23,000 definitions.[1]

1.2 The Lean theorem prover

The Lean project was launched in 2013 by Leonardo de Moura at Microsoft Research Redmond with the aim of creating a open source theorem prover that bridges the gap between automated and interactive theorem proving. This is an ongoing effort and its full potential for automation will only be achieved gradually. Nevertheless, Lean already comes with an interactive user interface, a flexible framework to support automation (Lean is its own *metaprogramming language*) and a rich API that allows Lean functionality to be embedded into other systems [22]. In addition to this, Lean has a small, efficient, trusted kernel written in C++ with support for multi-core machines and parallelism.

Lean 3, released in 2017, is the most stable version. It factored most of the core library out of the system repository. Based on this code, Mario Carneiro and Johannes Hölzl at Carnegie Mellon University launched the *mathlib* project: 'a community-driven effort to build a unified library of mathematics formalized in the Lean proof assistant' [27]. Its focus on classical mathematics and the welcoming community that gathered on Zulip has attracted many users, most of them with a background in mathematics, to Lean. Because

[1] Figures taken from the respective official websites: `mmlquery.mizar.org`, `www.isa-afp.org`, `math-comp.github.io`, and `leanprover-community.github.io`.

of this, *mathlib* has experienced a remarkable growth over the past three years. Some of the big-scale formalisation projects that depend on *mathlib* include the solution to the Cap Set problem [23], perfectoid spaces [26] and a proof of the independence of the Continuum Hypothesis [28].

Lean 4 is the latest release as of January 2021. It improves several aspects of the language, including more performant code generation and new metaprogramming capabilities. This project, however, will work solely with Lean 3, since the standard library has not yet been ported to Lean 4.

1.3 Objectives and contributions

The goals of this project are manifold. Firstly, I will give an overview of the system that can serve as an introduction to theorem proving in general, and to Lean in particular. Secondly, I will explore how Lean can be used to formalise mathematics at different levels. I intend to assess the expressiveness of the language, as a measure of how closely it resembles an informal proof on paper, as well as the state of automation, as a means to reduce the work needed in tedious proofs. We will also touch on Lean's status as a programming language. In doing so, we undertake a complete formalisation project that will state and prove results at the level of undergraduate mathematics. This will include International Mathematical Olympiad (IMO) problems and topics from group theory, such as composition series of groups and the Jordan–Hölder theorem.

I believe these initial aims have been completely met, resulting in the following contributions.

- A set of complete formalised solutions to four IMO problems from years 2014 and 2017, drawing from and extending the contents of *mathlib*.
- A careful development of the theory of composition series of groups, up to the Jordan–Hölder theorem for finite groups and a corollary. This was built from the existing theory of groups in *mathlib*, and often required filling gaps present in it.
- An extensive discussion (in the form of this report) of Lean and its library's features, through examples taken from the previous two formalisation endeavours. Special attention is paid to the axes of expressiveness, automation and usability.

In the process of achieving this, I have made a number of contributions to the *mathlib* project, in the form of pull requests to their GitHub repository[2]. Three of these added useful lemmas to different sections of the library, including results about square roots of natural numbers, finiteness of quotient types and images and preimages of group homomorphisms. Two more substantial pull requests added the second isomorphism theorem for groups. Lastly, there is a currently open pull request to add the formalised solution to question 1 in the 2014 IMO script to *mathlib*'s archive.

Chapter 2 is an introduction to Lean's features. Chapters 3 and 4 give an account of the two formalisation projects mentioned. Chapter 5 provides closing comments and ideas for further work.

[2] `https://github.com/leanprover-community/mathlib`

Chapter 2

Introduction to Lean

In this chapter, we review some of the basic features of Lean that will be needed in the further chapters. For a complete introduction to Lean, I have found the book *Theorem Proving in Lean* [24] extremely valuable. A shorter, more mathematics-oriented take can be found in *Mathematics in Lean* [25]. In addition to this, the community of Lean users on Zulip has always been very helpful in answering my questions.

2.1 Lean's type theory

Lean's kernel is based on dependent type theory. It uses a variant of the Calculus of Inductive Constructions that includes a sequence of non-cumulative type universes and inductive types. Lean's simple types will look familiar to those used to functional programming languages such as Haskell. One important distinction, however, is that Lean allows dependent types. In Haskell, terms can depend on types (for instance, the identity function `id : α -> α`) and types can depend on other types (via type constructors), but types cannot depend on terms. A dependent type theory, such as Lean's, observes this last case and loosens the difference between terms and types.

We will see some of Lean's features through an example. In Lean, the type of lists with elements of type `α` is denoted `list α`. When we write functions or theorems about lists, we would like these to apply to all lists and not just those with elements of a given type. This can be achieved with the dependent function type `Π`, also known as *pi* type or product type. Suppose `α` is some type and `β` is a function that takes an element of type `α` and produces a type (in Lean we write this as `β : α → Type`). Then `Π x : α, β x` is the type of functions `f` such that `f x` has type `β x` for each `x : α`. With this notation the familiar list constructor `cons`, which inserts a new element at the head of a list, has type `Π α : Type, α → list α → list α`. Because it would be very tedious to give the type argument `α` every time we call `cons`, Lean allows us to make this argument implicit by surrounding it with curly braces.

```
variable cons : Π {α : Type}, α → list α → list α
#check cons 2 [0,1] -- list ℕ
```

The `variable` keyword allows us to introduce new symbols into the working environ-

ment, which will be added as arguments to any definitions that mention them. The `check` command asks the system what the type of a given expression is. Note how we did not give the type `ℕ` explicitly as an argument to `cons`. Lean comes with a powerful elaborator that is in charge of instantiating these implicit arguments. Most of the time (as is the case in this example), the values of these arguments can be inferred from the arguments given explicitly or the expected return type of an expression. At times, however, the information that the elaborator is able to gather is insufficient to determine the value of an expression. In these cases, the expression is internally represented as a *metavariable* and the task of evaluating it is postponed until more information is available. In any case, for a complete declaration to be added to the environment, it must contain no unresolved metavariables. If we wish the elaborator to (attempt to) synthesise the value of an expression that we would otherwise be required to write explicitly, we can write an underscore in its place; this is known as a *hole*.

In Lean, every expression has a type, even `Type` itself. In fact, `Type` is an abbreviation for `Type 0`, which has type `Type 1`. Lean has a special type `Prop : Type`, where proposition terms live, that marks the beginning of the sequence of type universes. Each element of this sequence has two names: `Prop = Sort 0`, `Type 0 = Sort 1`, `Type 1 = Sort 2`, etc. Universe polymorphism is possible by introducing a universe variable (`universe u`) or writing `Type*`. The type `Prop` has two unusual properties. First, it is *impredicative*, which allows us to quantify over other universes and still remain inside `Prop`. Second, it is *proof-irrelevant*, which means that any two terms of a type in `Prop` are treated as definitionally equal. This allows the kernel to check proofs that depend on each other in parallel.

The main tool we have to construct new types are *inductive types* or, more generally, *inductive type families*. In fact, all types in Lean are built from the type universes, the `Π` type and inductive types. These are defined by providing a list of constructors that return elements of the new type. As an example, let us examine the definition of the list type. Note how the first argument α is given a name by moving it to the left of the colon.

```
1 universe u
2 inductive list (α : Type u) : Type u
3 | nil {} : list
4 | cons : α → list → list
```

We can read this as follows: a list of elements of type α (which may belong to any universe) is an empty list (the `nil` constructor) or is built up from an element of type α and another list (the `cons` constructor). The type α is bound at the top level, and cannot be changed inside the declaration. This is the reason we wrote `list` instead of `list α` in lines 3 and 4. Every inductive type comes with introduction and elimination rules. The first are simply its constructors; they are what allows us to build elements of the inductive type. The second allow us to use these elements by providing a principle of recursion, hence their name, *recursors*. In practice, we rarely need to use these recursors explicitly because Lean has a powerful pattern-matching mechanism that will compile simpler, more readable definitions and proofs into ones written in terms of recursors under the hood. This is known as the *equation compiler*. Thanks to this, we can painlessly write a function that appends two lists.

```
def append {α : Type*} : list α → list α → list α
| nil        t := t
| (cons a s) t := cons a (append s t)
```

When an inductive type has a single constructor, say `mk`, which takes arguments `a` and `b`, we can write `⟨a, b⟩` instead of `mk a b`. This is known as an *anonymous constructor*. Types with only one constructor can instead be declared as structures, which allows the naming of each field and automatically generates the respective projections. Lastly, suppose we have a term `x` of type `foo`, and a function `foo.bar` in the `foo` namespace that takes a term of type `foo` as its first explicit argument. Then we can write `x.bar` instead of `foo.bar a`.

2.2 The tactic system

As we mentioned earlier, Lean has powerful automation possibilities that aid the user in the process of constructing proofs. These are realised by the *tactic* system. A tactic can be seen as a command that takes care of building part of a proof (sometimes the entire proof) automatically. There are many tactics included in Lean by default and *mathlib* has many of its own. Most importantly though, new tactics can be written by the user as needed by using Lean's metaprogramming capabilities. We will not go into the details of how to create your own tactics in this text, but we will get a sense of their power.

In order to use tactics, one needs to enter *tactic mode*, which can be done using a `begin...end` block or with the `by` keyword. Overall, for each construct in a non-tactic proof there is a corresponding tactic. For example, in tactic mode we write `intros p q hp hq` instead of `assume (p q : Prop) (hp : p) (hq : q)`. When in tactic mode, if we are using a text editor with Lean support such as Emacs or VSCode, a panel will show us what our goal is and what assumptions and variables are available in the local context.

Two of the most ubiquitous tactics are `rewrite` and `simp`. The `rewrite` tactic (usually abbreviated `rw`) helps us work with equality by applying given substitutions to the first section of the goal that matches the pattern. The lemmas to use in these substitutions are given in a list surrounded by brackets. Rewriting is a rather surgical tool, where one has to specify every step of the process. This can easily become impractical and, in those cases, the simplifier (invoked with `simp`) will reduce the amount of typing and library searching needed. A number of theorems in the library are tagged with the `[simp]` attribute, and the simplifier tries to use these to rewrite the goal into a canonical form. If it is successful, both sides of the goal equation reduce to the same form, and it finishes the proof for us. We can provide `simp` with extra theorems if needed by listing them as for `rewrite`.

2.3 Type classes

The last feature of Lean we will discuss are *type classes*. These constructions, inspired by Haskell's eponymous feature, are used to associate operations and predicates to existing types. Type classes are heavily used by *mathlib* because they are a natural way to organise mathematical structures and their hierarchy. As an example, we can create a type class for types that support some kind of addition operation. In this case, it is convenient to introduce the usual `+` notation.

```
class has_add (α : Type*) :=
(add : α → α → α)

def add {α : Type*} [has_add α] : α → α → α := has_add.add
notation a ` + ` b := add a b

instance nat_has_add : has_add ℕ := ⟨nat.add⟩
```

We used square brackets around `has_add α` in line 4 to indicate that we would like this argument to be derived by the type class inference mechanism. Once we have defined a class, we can populate it with instances, which in this case will be the types with an addition operation. Line 7 allows us, among other things, to use `+` with natural numbers to mean addition. When we input the `+` symbol, Lean knows to look for an instance of the `has_add` class which will define what addition means in each case. This task, which is carried out by the elaborator, is known as *type class resolution*.

The power of type classes extends way beyond operator overloading, however. An important property is that we can require instances in definitions, theorems and instance declarations themselves. This allows us to, for example, define addition component-wise for the product of two types that support addition.

```
instance prod_has_add {α β : Type*} [has_add α] [has_add β] :
  has_add (α × β) :=
⟨λ p q, ⟨p.1 + q.1, p.2 + q.2⟩⟩
```

We can also have inheritance between classes, not unlike inheritance in imperative programming languages. A class that extends a previously defined class will inherit its associated operations and predicates, as well as introduce some of its own. An example of this is the definition of an additive semigroup in the *mathlib* source code.

```
class add_semigroup (G : Type u) extends has_add G :=
(add_assoc : ∀ a b c : G, a + b + c = a + (b + c))
```

Note that, since we are extending `has_add G`, we are allowed to use the `+` notation. It is now easy to see how type class inheritance can be used to express the hierarchy of different algebraic structures. For instance, the class `ring` could extend the class `add_comm_group`, which would instantly make all theorems about commutative groups available to rings.

One more thing that type classes allow is *coercion*. In Lean, we can have three types of coercion from a family of types: to another family of types, to the class of sorts and to the class of function types. The first one is the most common. It allows us to treat natural numbers as integers, integers as rationals and rationals as reals, among other things. The remaining two are useful when dealing with algebraic structures that are built on top of a carrier type and morphisms between these. When a term of the wrong type is provided in some expression, Lean will look for a coercion to the expected type, and signal a type error if it does not find one. At times, it is necessary to explicitly tell Lean that we want a coercion to take place. In those cases we can use the lift operator. Each of the three types of coercions has their own lift operator: `↑`, `↥` and `⇑`, respectively. Sometimes, we even need to explicitly write what type we want to coerce into. For instance, `1 : ℤ` coerces `1`, which by default a natural number, into an integer.

Chapter 3

Formalising solutions to IMO problems

We begin our exploration of Lean with what is a traditional exercise for newcomers: formalising International Mathematical Olympiad (IMO) questions. These provide a good introduction to Lean and *mathlib* because they tend to involve a fair share of basic types, but, at the same time, pose a bigger challenge than, say, writing solutions to textbook exercises. IMO questions are historically divided into four categories: algebra, number theory, combinatorics and geometry. The first two are arguably the best suited for formalisation using existing *mathlib* types and results. Formalising results in combinatorics often requires defining new constructs in the form of inductive types. As for geometry problems, they can be solved either algebraically or synthetically. The first option produces heavily computational proofs that are seldom legible to a human, but which have great potential for automation [8]. Synthetic proofs, on the other hand, mirror traditional geometrical reasoning and can often be read by humans. Automation of synthetic proofs can be achieved through a number of techniques, such as the *area method* [17] or the *full-angle method* [13]. Tackling either of these using Lean could easily constitute a project of its own.

For simplicity, we will only focus on questions categorised as algebra and number theory. In this chapter, I present formalised solutions to four problems from two IMO scripts (2014 and 2017). They are mostly adapted from the model solutions, which can be found on the IMO official website[1], with minor generalisations where appropriate. The end results are rather lengthy and not every step in the process provides insight into Lean's capabilities. Therefore, I will only provide an overview of the main points in each question. In each of them, we will see how to work with and prove things about different types in Lean. The first two questions deal mostly with sequences of integers, the third with real valued functions, and the fourth with multivariable polynomials with integer coefficients. The following table provides some statistics about the formalised solutions:

[1] `https://www.imo-official.org/problems.aspx`

Problem	Definitions	Lemmas	Lines of code
2014 Q1	1	11	146
2017 Q1	2	25	521
2017 Q2	2	13	185
2017 Q6	10	16	563
Auxiliary	2	65	669

3.1 Sequences of integers: 2014 question 1

We begin with a problem about sequences of integers.

> Let $a_0 < a_1 < a_2 < \cdots$ be an infinite sequence of positive integers. Prove that there exists a unique $n \geq 1$ such that
> $$a_n < \frac{a_0 + a_1 + \cdots + a_n}{n} < a_{n+1}.$$

One sensible way one might formalise a infinite sequence of integers is as a function from ℕ to ℤ. We will declare our sequence a as a variable.

```
variable (a : ℕ → ℤ)
theorem imo2014q1 (hpos : ∀ n, 0 < a n) (hinc : ∀ n, a n < a (n + 1)) :
  ∃! n : ℕ+, (a n : ℚ) < (↑∑ i : fin (n + 1), a i) / n ∧
  (↑∑ i : fin (n + 1), a i : ℚ) / n ≤ a (n + 1) := sorry
-- from src/imo/imo2014_q1.lean
```

There are few things to unpack from the type signature of the theorem. The first two arguments (`hpos` and `hinc`) are the defining assumptions for a in the question statement. What follows is what we want to show. The notation `∃! x : α, p x`, read as 'there exists a unique `x` of type `α` such that `p x`', is shorthand for `∃ x : α, p x ∧ ∀ y : α, p y → y = x`. In this case, the type `α` is `ℕ+` which is the type of positive natural numbers. The condition that our unique `n` needs to satisfy is made up of two inequalities of rational numbers. Because all terms involved are integers, we need to explicitly coerce them into type `ℚ` or else the division operation (`/`) will be integer division. Moreover, we need to use the lift operator (`↑`) in the numerator of our fraction lest the coercion to `ℚ` happens only after division. Lastly, we wrote `sorry` to let Lean know that we do not intend to provide a proof just yet.

There are two types in this declaration that require further attention: `ℕ+` and `fin (n + 1)`. Both are examples of what Lean calls a subtype. If we have a type `α` and a predicate `p : α → Prop`, we can construct a new type that contains only those terms `x : α` such that `p x` is true. This is written `{ x // p x }`[2]. Internally, a subtype is nothing more than a structure with two fields: `val`, of type `α`, and `property`, of type `p val`. With the custom notation, `ℕ+` is defined as `{ n : ℕ // 0 < n }`, while for any `n : ℕ` the type `fin n` is defined as `{ i : ℕ // i < n }`. The latter type is special for yet another reason: it is finite. In Lean, finite types are members of the `fintype` type class. This allows, among other things, to take sums over them. This is precisely what is done in lines 3 and 4 with the `∑` notation: `∑ i : fin (n + 1), a i` is the equivalent of the informal $a_0 + a_1 + \cdots + a_n$.

[2] The notation `{ x | p x }` is reserved for sets.

In order to solve this question, we will use an auxiliary sequence that will transform our goal into an inequality of integers, instead of rationals.

```
def d (n : ℕ+) : ℤ := (∑ i : fin (n + 1), a i) - n * (a n)

lemma d_one : d a 1 = a 0 :=
show a 0 + (a 1 + 0) - 1 * a 1 = a 0, by simp
-- from src/imo/imo2014_q1.lean
```

This lemma showcases the `show` syntax, which is used when we want to make the type of the goal explicit in the proof. In this case, we use it to unfold the definition of `d` manually. Even if the statement of the lemma seems different from the goal we wrote after `show`, the two are definitionally equal. We use `show` as a hint to Lean that we want to work with the expanded definition, instead of `d a 1`. The parentheses are needed because the sum notation associates to the right. If they were not there, the goals would not be definitionally equal and `show` would fail. In addition to this, line 4 shows the power of the simplifier in proving fairly trivial equalities in arbitrary rings. For the next lemma, however, the simplifier is not enough.

```
lemma ddes (hinc : ∀ n, a n < a (n + 1)) (n : ℕ+) : d a (n + 1) < d a n :=
lt_of_sub_neg $
  calc ((∑ i : fin (n + 2), a i) - (n + 1) * a (n + 1))
         - ((∑ i : fin (n + 1), a i) - n * a n)
       = ((∑ i : fin (n + 1), a i) + a (n + 1) - (n + 1) * a (n + 1))
         - ((∑ i : fin (n + 1), a i) - n * a n) : by simp [fin.sum_univ_cast_succ]
  ... = n * (a n - a (n + 1))                   : by ring
  ... < 0
     : mul_neg_iff.mpr (or.inl ⟨by simp, sub_neg_of_lt (hinc n)⟩)
-- from src/imo/imo2014_q1.lean
```

This shows that if `a` is strictly increasing, then `d a` is strictly decreasing. The lemma `lt_of_sub_neg` says that it is enough to show that `d a (n + 1) - d a n < 0`. The dollar sign in line 2 is analogous to Haskell's `$`; writing `f $ ...` is the same as `f (...)`, so it saves us closing one bracket at the end. What follows is known as a `calc` block. We need to prove `d a (n + 1) - d a n < 0`, so we do it through a series of equalities and inequalities. Each equality is proved after the colon following it. Internally, the block is compiled to the appropriate applications of the relevant transitivity properties. Usually, the left-hand side of the first equality would be `d a (n + 1) - d a n` and the right-hand side of the last one should be `0`. However, in this case we take the chance to unfold the definition of `d` as we did with `show` before. Note that the first equality is proved by `simp`, but for the second we invoke the `ring` tactic. This is a powerful tactic implemented by Mario Carneiro based on [12] that is capable of proving most equalities following from the (commutative) ring axioms. It does a lot more than `simp`, since, for instance, it can cancel terms that are not initially adjacent.

Let us have a look at one more instructive example taken from this formalised solution. It is a general result about descending functions from ℕ+ to ℤ.

```
variables {f : ℕ+ → ℤ} (hdes : ∀ n, f (n + 1) < f n)
include hdes
```

```
3
4  lemma lt_of_lt_of_des {n m : ℕ+} (hnm : n < m) : f m < f n :=
5  begin
6    have : ∀ k : ℕ, f (n + ⟨k + 1, nat.succ_pos k⟩) < f n,
7    { intro k, induction k with k ih,
8      { exact hdes n },
9      apply lt_trans (hdes _) ih },
10   convert this (↑(m - n) - 1),
11   have : 1 ≤ ↑(m - n) := (pnat.coe_le_coe 1 _).mpr (pnat.one_le _),
12   simp [nat.sub_add_cancel this, pnat.add_sub_of_lt hnm],
13 end
14 -- from src/imo/imo2014_q1.lean
```

Because the lemmas following this one in the solution file also deal with descending sequences such as `f`, I declared `f` and `hdes` as variables to prevent repeating their types in every lemma. Line 2 includes the assumption `hdes` in all the lemmas following it. This is needed, since the type signature of `lt_of_lt_of_des` makes no mention of `hdes`, and yet we intend to use it in the proof. In line 6, we introduce an anonymous temporary subgoal, which is proved in lines 7–9. The keyword `this`, used in lines 10 and 12, then stores the result of the most recent anonymous subgoal.

The first subgoal is set up to be proved by induction on `k`. Note how in its statement, `k+1` is made into type `ℕ+` using the anonymous constructor and a proof that it is positive. We use the induction tactic in line 7, which creates two goals: the base case (`f (n + ⟨0 + 1, _⟩) < f n`) and the inductive step (`f (n + ⟨k.succ + 1, _⟩) < f n`). In the absence of an explicit induction principle, or recursor, the induction tactic will use the one automatically generated for the type. The `with k ih` part simply names the variable and the induction hypothesis for the inductive step case. The first goal is closed in line 8, using the included assumption that `f` is descending, and the second is closed in line 9. The `exact` tactic attempts to close the goal with the provided term. The `apply` tactic is very similar, except it allows holes (_) for parts of the term that are to be determined, and creates new goals for them. In the case of line 9, no further goal is created even if we used a hole because the resulting goal is true by definitional equality.

Line 10 applies our subgoal from line 6 to the main goal, by using the `convert` tactic. This tactic allows us to use a term that is not definitionally equal to the goal, and creates new goals that show that the differences between the used term and the goal are provably equal. For instance, in this case `this (↑(m - n) - 1)` has type `f (n + ⟨↑(m - n) - 1 + 1, _⟩) < f n`, whereas our goal is `f m < f n`. After using `convert`, the only thing left to prove is that `m = n + ⟨↑(m - n) - 1 + 1, _⟩`, which is done in lines 11 and 12.

The rest of the solution is mostly straightforward, and does not particularly showcase anything that we have not already seen. Once we defined the auxiliary sequence `d a`, we can translate the question to finding a unique `n : ℕ+` such that `0 < d a n` and `d a (n + 1) ≤ 0`. But we have already seen that `d a` is strictly descending, and the result follows with the help of a few more lemmas about descending sequences.

3.2 Recurrence relations: 2017 question 1

The next problem chosen deals with a recurrence relation on the natural numbers.

> For each integer $a_0 > 1$, define a sequence $a_0, a_1, a_2, \dots$ by
>
> $$a_{n+1} = \begin{cases} \sqrt{a_n}, & \text{if } \sqrt{a_n} \text{ is an integer,} \\ a_n + 3, & \text{otherwise} \end{cases} \qquad \text{for every } n \geq 0.$$
>
> Determine all values of a_0 for which there is a number A such that $a_n = a$ for infinitely many values of n.

Defining such a sequence in Lean begs for a recursive definition, which can be easily written down with the help of the equation compiler.

```
def a (a₀ : ℕ) : ℕ → ℕ
| 0     := a₀
| (n+1) := let k := sqrt (a n) in
           if k * k = a n then k else a n + 3

def periodic (a₀ : ℕ) : Prop := ∃ A, ∀ n, ∃ m, n ≤ m ∧ a a₀ m = A
-- from src/imo/imo2017_q1.lean
```

We use a `let` binding in line 3 to avoid repeating `sqrt (a n)` in the definition. Line 4 uses the `if ... then ... else` syntax, which is familiar from other programming languages. In Lean, this can only be used when the proposition that we give as a condition is a member of the `decidable` type class. In this case, `k * k = a n` is an equality of natural numbers (`sqrt n` returns the largest natural number `k` such that `k * k ≤ n`), which is decidable. Because of this, our definition of `a` is entirely computable, and we can write `#eval a 4 6` to get `17`, the sixth term in the sequence starting with $a_0 = 4$. This is a great example of how Lean is not only a proof assistant, but also a programming language: we are defining a function that can be compiled to bytecode and, at the same time, proving its properties. In line 6 we provide a definition of what it means for a starting value `a₀` to result in a periodic sequence. The question asks us to find all `a₀` such that `periodic a₀`.

The solution to this problem uses a great deal of arithmetic modulo 3. In *mathlib*, there are two ways of expressing this. The first is modular equality, written `a ≡ b [MOD n]`, which is defined as `a % n = b % n`, where `%` is the modulo operator. The second is using the `zmod n` type, which is closely related to the type `fin n` that we saw in the previous question. In fact `zmod n` is defined as `fin n` whenever `n` is not `0`, while `zmod 0` is defined to be `ℤ`. The crucial difference between `zmod` and `fin` is that the former is equipped with a commutative ring structure, i.e., it is a member of the `comm_ring` type class. The structure is that of the integers modulo `n`. If we are given two natural numbers `a` and `b`, then an equivalent way of saying that `a` is congruent to `b` modulo `n` is writing `(a : zmod n) = b`. The two ways of formalising this concept are related by the lemma `zmod.eq_iff_modeq_nat` which says that `∀ (n : ℕ) {a b : ℕ}, (a : zmod n) = b ↔ a ≡ b [MOD n]`. There are reasons to work with both representations throughout the proof. The modular equality statement has the benefit of staying within the `ℕ` type, which avoids dealing with coercions. On the other hand, `zmod` has a ring structure (which `ℕ` does not), and it is a finite type, which enables powerful automation.

As an example of how these two representations can be used together, consider the following lemma showing that a number congruent to 2 module 3 cannot be a perfect square.

```
lemma not_square_of_two_mod_three {n : ℕ} (h : n ≡ 2 [MOD 3]) :
  ¬∃ m, m * m = n :=
begin
  rintro ⟨m, rfl⟩,
  have : ¬∃ (m : zmod 3), m * m = 2 := by dec_trivial,
  apply this ⟨m, _⟩,
  norm_cast,
  rwa ←zmod.eq_iff_modeq_nat at h,
end
-- from src/imo/imo2017_q1.lean
```

In Lean, the negation of a proposition `p` is defined as `p → false`. Therefore, to prove `¬∃ m, m * m = n` we introduce an assumption `∃ m, m * m = n` and try to reach a contradiction. Line 4 both introduces this assumption and destructures it using pattern matching. That way, we obtain an `m` such that `m * m = n`. The latter property is not given a name, instead `rfl` is used. This notation is possible thanks to the `rintro` tactic and, instead of adding the assumption `m * m = n` to the context, it uses it to rewrite the type of every assumption in the context that contains `n`. In this case, `n` is present in `h`, so after line 4 the assumption `h` becomes `m * m ≡ 2 [MOD 3]`. The crucial part of this lemma comes in line 5. Here we prove a subgoal which is equivalent to our target, except it is expressed in terms of the `zmod 3` type. Because this is a finite type, Lean can simply check each of the three elements to confirm that no `m` exists such that `m * m = 2`. This is done by the `dec_trivial` tactic. What follows transfers this result back to the modular equality language. This is a great example of tedious proof automation, which would be even more valuable if we were working in `zmod n` for some big `n`.

This lemma already rules out a_0 from begin congruent to 2 modulo 3, for in that case no perfect square is reached and the sequence is forever increasing. The remainder of the solution works to eliminate the case where a_0 is congruent to 1 modulo 3. The formalisation was not as simple as for the previous question, partly because the problem itself was harder, and partly due to the constant interchange between types `ℕ` and `zmod 3`.

3.3 Functional equations: 2017 question 2

For the third problem we have a functional equation over the real numbers.

> Let $\mathbb{R}$ be the set of real numbers. Determine all functions $f : \mathbb{R} \to \mathbb{R}$ such that, for all real numbers x and y,
>
> $$f(f(x)f(y)) + f(x+y) = f(xy).$$

Due to its algebraic nature, this question was by far the easiest to formalise in Lean. Most of the work involved rewriting the equation above under different conditions, for which `calc` blocks are very well suited. After a number of reduction steps, it turns out that the only functions satisfying the initial equation are the constant function at 0, the func-

tion sending x to $x-1$ and its negation. The complete solution can be found in the file `src/imo/imo2017_q2.lean`.

3.4 Multivariate polynomials: 2017 question 6

We finish with a more substantial question, that uses a range of different types. Formalising this solution was one of the greatest challenges of this project. It involved getting familiar with various distant areas of the library, and it exposed one of the biggest limitations of the type class system.

> An ordered pair (x,y) of integers is a primitive point if the greatest common divisor of x and y is 1. Given a finite set S of primitive points, prove that there exists a positive integer n and integers $a_0, a_1, \ldots, a_n$ such that, for each (x,y) in S, we have:
>
> $$a_0x^n + a_1x^{n-1}y + a_2x^{n-2}y^2 + \cdots + a_{n-1}xy^{n-1} + a_ny^n = 1$$

First of all, we define what a primitive point is in Lean, along with a useful lemma.

```
1 def primitive_point (p : ℤ × ℤ) : Prop := int.gcd p.fst p.snd = 1
2
3 lemma of_mul_primitive {p : ℤ × ℤ} {n : ℤ} (h : primitive_point (n • p)) :
4   primitive_point p ∧ (n = 1 ∨ n = -1) := ...
5 -- from src/imo/imo2017_q6.lean
```

Here • is the symbol of scalar multiplication, resulting from seeing $\mathbb{Z}\times\mathbb{Z}$ as a $\mathbb{Z}$-module. In writing the proof of `of_mul_primitive`, I encountered the diamond problem for the first time. In order to be able to write • for scalar multiplication, Lean needs to derive an instance of `has_scalar ℤ (ℤ × ℤ)`. Because the type class graph is far from being a tree[3], there is usually more than one instance derivation path that can be taken. In this case, `has_scalar ℤ (ℤ × ℤ)` is derived from an instance `has_scalar ℤ ℤ`, which in turn is derived from other instances, etc. The problem arose when I attempted to use the following lemma in `alegbra.module.basic`.

```
1 lemma neg_smul {R : Type u} {M : Type w} [ring R] [add_comm_group M] [module R M]
2   (r : R) (x : M) : -r • x = -(r • x)
```

In our context, `R` is `ℤ` and `M` is `ℤ × ℤ`. This lemma requires three instances. The first one can be provided directly with `int.ring`. There are several paths that can be taken for the second one. The type class resolution system tries instances in the order that they are imported, and only seldom does it actually follow the path that one would expect. Most of the time, this is not a problem because many of these paths end up with definitionally equal instances. If this is not the case, we have found ourselves a 'diamond' that does not 'commute'.

The issue, in our case, comes when Lean tries to derive the `module ℤ (ℤ × ℤ)` instance. The two prime candidates differ in whether they use `semiring.to_semimodule` or `add_`

[3] Figure 1 in [27] shows part of this graph.

`comm_group.int_module`. Either one results in a valid instance, but only the first one results in a scalar multiplication that is definitionally equal to the one we use in `of_mul_primitive`. Unfortunately, the second option is explored by the type class search first, which renders an attempt to rewrite using `neg_smul` unsuccessful. This can be discovered by typing `set_option trace.class_instances true` in the file, which will cause every line after it using type class resolution to report a log of its search. The simplest solution to our problem is to redeclare the instance we need at the top of our file by writing `attribute [instance] semiring.to_semimodule`. This will cause the graph search to explore this path first, resulting in a instance of `module ℤ (ℤ × ℤ)` compatible with the `has_scalar ℤ (ℤ × ℤ)` instance.

Problems arising from diamonds are not very common, mostly because a lot of effort is put in the design of type classes to avoid them, but when they do arise it can be extremely hard to spot exactly where the issue is. If one does not have a good knowledge of the library, the best strategy for finding what instance might be causing the problem is to go through the search logs, which are, to say the least, unfriendly. This is an aspect Lean needs to improve. There is little documentation on how type class inference works and, though most of the time it is a very helpful feature, it can cause unexpected problems that are very difficult to solve.

Leaving type classes aside, let us see how one might formalise the statement of the question. We are given a finite set of primitive points and need to find some two-variable polynomial homogeneous polynomial that evaluates to 1 at every point in the set. Luckily, *mathlib* already comes with all the types and predicates we need, with the exception of `primitive_point` defined above. The type of finite sets on `ℤ × ℤ` is written `finset (ℤ × ℤ)`. Multivariate polynomials are specified with an index type, where each term represents a variable, and a commutative ring, whose elements are the coefficients. The type of two-variable polynomials over the integers can be written as `mv_polynomial (fin 2) ℤ`, where we could have chosen any two-element type instead of `fin 2` to index over our variables. We can evaluate a polynomial by giving a value to each of its variables, which amounts to providing a function `σ → R`. If `f` is such a function and `φ : mv_polynomial σ R`, then `φ` evaluated at `f` is written `eval f φ`. The last ingredient needed to formalise the statement of the question is the definition of a homogeneous polynomial, i.e., a polynomial in which every monomial has the same degree. If this degree is `n`, we write `is_homogeneous φ n` to signify this.

```
1  def to_val (p : ℤ × ℤ) : fin 2 → ℤ
2  | ⟨0, _⟩   := p.1
3  | ⟨1, _⟩   := p.2
4  | ⟨n+2, h⟩ := absurd h (by simp)
5
6  theorem imo2017q6 (S : finset (ℤ × ℤ)) (hS : ∀ t ∈ S, primitive_root t) :
7    ∃ (φ : mv_polynomial (fin 2) ℤ) (n : ℕ), 0 < n ∧ is_homogeneous φ n ∧
8      ∀ t ∈ S, eval (to_val t) φ = 1 := sorry
9  -- from src/imo/imo2017_q6.lean
```

First we need to define how to evaluate a polynomial at a point of `ℤ × ℤ`. This is done with the `to_val` function, which creates a valuation function from a point `p`, sending `0` to the first coordinate and `1` to the second. Line 4 is needed, or else the equation compiler will

flag that we did not exhaust all cases. We can then use `to_val` to evaluate our polynomial at every point in `S`.

The formalisation of this solution was an arduous job and not much would be relevant to present here. Perhaps an interesting step was the one generalising Bézout's identity to lists of integers. It is another good example of using Lean as both a functional programming language and a theorem prover. The library already has a computable definition of Bézout's identity for two integers. This is achieved in the form of two functions `gcd_a` and `gcd_b` that take two arguments each, and satisfy the equation `gcd x y = x * gcd_a x y + y * gcd_b x y`. Using these, our goal is to define a function `bezout_factors` that, given a list $a_1, \ldots, a_n$ of integers, computes another list $x_1, \ldots, x_n$ of integers such that

$$\gcd\{a_1, \ldots, a_n\} = x_1 a_1 + \cdots + x_n a_n.$$

Given `l : list ℤ`, we can take its greatest common divisor as `l.foldr gcd 0`. The key to computing the desired coefficients is the realisation that, if S is a finite set of integers , then $\gcd(\{a_1\} \cup S) = \gcd\{a_1, \gcd S\}$. Using the original Bézout identity, we can find coefficients x_1 and x_2 such that

$$\gcd\{a_1, \gcd\{a_2, \ldots, a_n\}\} = x_1 a_1 + x_2 \cdot \gcd\{a_2, \ldots, a_n\},$$

giving us the inductive step. In Lean, this translates to the following recursive definition.

```
1  def bezout_factors : list ℤ → list ℤ
2  | []        := []
3  | (x :: tl) := let g := tl.foldr gcd 0 in
4    gcd_a x g :: (bezout_factors tl).map ((*) (gcd_b x g))
5
6  lemma bezout_eq_gcd : Π (l : list ℤ),
7    (l.zip_with (*) (bezout_factors l)).sum = l.foldr gcd 0
8  | []        := by simp
9  | (x :: tl) := begin
10   simp [bezout_factors, zip_with_mul_map_mul_right, sum_map_mul_left],
11   show _ + _ * (map id _).sum = _,
12   rw [map_id, bezout_eq_gcd tl, gcd_eq_gcd_ab, mul_comm (gcd_b _ _)],
13 end
14 -- from src/imo/bezout.lean
```

Since `gcd_a` and `gcd_b` were originally computable, so is our function. We also prove that our algorithm is correct. This proof looks a lot like a recursive definition. In fact it *is* one, only its return type is a `Prop`. The result follows from the definition, the correctness of `gcd_a` and `gcd_b` and a couple of lemmas about list operations. The use of the `show` syntax here is a consequence of the limitations of the `rewrite` tactic. The goal state after `simp` has the form `_ + _ * (map (λ a, a) _).sum = _`. Of course, this map is not actually doing anything, since the function mapped is the identity. The lemma `map_id` states that `map id l = l` for any list `l`. However, in order to use it, we must manually change the goal to be written in terms of the `id` function. We can do this using `show` because `id` is definitionally equal to `λ a, a`.

Chapter 4

Composition series and the Jordan–Hölder theorem

So far, we have used Lean and *mathlib* to formalise results about already existing types: sequences of integers, functions on the real numbers and multivariate polynomials. In this chapter, we will create new types and prove results about them. The task of defining new objects in a theorem prover is very different in flavour to formalising proofs. Back in Section 2.1, we saw how terms of type `Prop` are treated as definitionally equal and, once a theorem is proved, Lean never has to revisit its proof – theorems are a sort of black box. On the other hand, definitions can, and often must, be unfolded; especially if we are trying to prove something about the objects they define. This means that, even though any proof is as good as another (leaving aside matters of performance and elegance), the wrong definition may render some results harder to prove at best, and at worst impossible.

The theory chosen is that of composition series of groups, which is traditional in an advanced undergraduate course in group theory. We will only give a brief introduction to the subject. For a complete treatment, see for example Chapter 5 of [18]. In short, a normal series (sometimes subnormal series) of a group G is a sequence of subgroups

$$1 = G_0 \leq G_1 \leq \cdots \leq G_{n-1} \leq G_n = G,$$

where G_{i-1} is normal in G_i for each $i \in \{1, \ldots, n\}$. We call the quotient groups G_i/G_{i-1} the factors of the normal series. A composition series is a normal series whose factors are all simple and nontrivial. We say that a group is simple if its only normal subgroups are the trivial subgroup and itself. It can be proven – in fact, we will prove it – that every finite group has a composition series. The Jordan–Hölder theorem states that any two composition series for a group G have the same factors, up to isomorphism and permutation. For simplicity, we will only formalise a proof of the special case where G is finite, which proceeds by induction on the order of G.

The first two sections introduce two attempts at defining composition series. We will see which of the two is best suited for *mathlib*, and use it to formalise the theory up to the proof of the Jordan–Hölder theorem. In the last section, we use the proven results to derive the fundamental theorem of arithmetic as a corollary to Jordan–Hölder. The following table shows an overview of the size of the development:

File	Definitions	Lemmas	Lines of code
`normal_series.lean`	14	10	270
`normal_embedding.lean`	7	25	140
`arithmetic.lean`	3	13	185
Auxiliary	19	88	821

4.1 Out of a textbook: a first definition

Since our intentions are to expand the contents of *mathlib*, we will be using the existing framework in our work. The library already contains a modest number of basic group theoretic definitions and results: the definition of a group (as a type class), the type of subgroups of a group, the definition of a normal subgroup, and various results about group homomorphisms (including the first isomorphism theorem). Since we will be using subgroups extensively, it is worth examining how they are defined.

```
1  structure subgroup (G : Type*) [group G] :=
2  (carrier : set G)
3  (one_mem : (1 : G) ∈ carrier)
4  (mul_mem : ∀ {a b : G}, a ∈ carrier → b ∈ carrier → a * b ∈ carrier)
5  (inv_mem : ∀ {x : G}, x ∈ carrier → x⁻¹ ∈ carrier)
6
7  variables {G : Type*} [group G]
8  instance : has_coe (subgroup G) (set G) := ⟨subgroup.carrier⟩
9  instance : has_mem G (subgroup G) := ⟨λ g H, g ∈ (H : set G)⟩
10 instance : has_coe_to_sort (subgroup G) := ⟨_, λ H, { g // g ∈ H }⟩
11
12 class normal (N : subgroup G) :=
13 (conj_mem : ∀ (n : G), n ∈ N → ∀ (g : G), g * n * g⁻¹ ∈ N)
```

This is not the definition found in *mathlib* word by word, but rather it is one that compiles to the same primitives. A subgroup simply consists of a set of elements of `G` that satisfies the closure properties given in lines 3–5. As such, there is a natural coercion from `subgroup G` to `set G`, which enables a `has_mem` instance and a coercion to the class of sorts. Normality is then defined as a type class, which just includes an extra closure property. Equipped with these, we may produce a first attempt at defining normal series in Lean.

Given the definition presented above, it is natural to view a normal series of G as a list of subgroups of G that satisfies a number of conditions. This translates to Lean in a fairly direct way.

```
1  structure normal_series (G : Type*) [group G] :=
2  (series : list (subgroup G))
3  (nonempty : series ≠ [])
4  (head : series.head = ⊥)
5  (last : series.last nonempty = ⊤)
6  (normal : ∀ (i : ℕ) (h : i + 1 < series.length),
7    series.nth_le i (nat.lt_of_succ_lt h) ◃ series.nth_le (i + 1) h)
8  -- from src/jordanholder/normal_series2.lean
```

Previous formalisations of normal series in other proof assistants follow a more or less

analogous approach – see for instance [14] in Mizar[1], [21] in Isabelle and [20] in Coq. In essence, given a type `G` (in any type universe) with a `group` instance, we have defined `normal_series G` to be a structure with five fields. The first is the only one that contains any data, namely a list of subgroups of `G`. The remaining fields have types that are propositions, which require the list to be nonempty, its head to be the trivial subgroup (written `⊥`), its last element to be the entire group (the universal subgroup is written `⊤`)[2] , and each subgroup in the list to be normal in the next. At first glance, this definition seems sensible – it almost mirrors the textbook one word by word. Nevertheless, it has two major limitations: it needs an intermediate definition of normality, and it does not use the full power of Lean's type theory.

As we saw earlier, *mathlib* already comes with a definition of what it means to be a normal subgroup. This however only applies to a subgroup `N : subgroup G` being normal in `G`. In informal mathematics, if we have a group G and two subgroups H and K of G with the property that $H \subseteq K$, we usually think of H as both a subgroup of G and a subgroup of K. In Lean, H and K would have type `subgroup G` and one would write `H.normal` for the proposition that `H` is normal in `G`. Although there is a coercion from the type `subgroup G` to the class of sorts, and it inherits an instance of the `group` type class, there is no immediate way to coerce `H : subgroup G` with `H ≤ K` into the type `subgroup K`. Because of this, and since the definition of N being normal in $H \leq G$ is straightforward, I provided my own definition of normality as `normal_in`. This then allows us to use the usual symbol to denote normality (`◃`), by defining it as infix notation for `normal_in`.

```
def normal_in {G : Type*} [group G] (N H : subgroup G) : Prop :=
N ≤ H ∧ ∀ n, n ∈ N → ∀ h, h ∈ H → h * n * h⁻¹ ∈ N
infix ` ◃ `:50 := normal_in
-- from src/jordanholder/normal_series2.lean
```

This solution comes at price: it does not give us access to the already existing results about normal subgroups, which are essential to further develop the theory. For instance, we cannot construct the quotient H/N from the fact `normal_in N H` without redefining quotient groups entirely. The only sensible way forward would be to relate `normal_in` to the already existing definition of normality, but this boils down to the problem of coercing from `subgroup G` to `subgroup H`. We will see in Section 4.3.2 that this problem can be solved in a satisfactory way, but it would clutter the definition and detract from its biggest appeal: its simplicity.

The second, and perhaps most important, limitation of our definition is that it does not utilise the full capabilities of Lean's type theory. By using the homogeneous `list` type, the elements of our `normal_series` are forced to have type `subgroup G`. As we have just seen, this conflicts with the definition of normality that *mathlib* offers. Perhaps it would be beneficial to have each element `H` of the series be of type `subgroup H'`, where `H'` is the next element. This is not possible using lists, but it could easily be achieved with an inductive type (provided that we define the sequence from right to left). In the next section,

[1] The development in [14] skips the definition of normal series altogether, and directly defines composition series. Their definition is still a sequence of subgroups, with the extra defining assumptions of a composition series.

[2] This notation is borrowed from the `bounded_lattice` type class, of which `subgroup G` is a member. The symbols `⊤` and `⊥`, read 'top' and 'bottom', denote the greatest and least elements, respectively.

we will see how this approach can lead to a definition that is far more natural in a theorem prover, even if one could never find it in a textbook.

4.2 Change in perspective: a second definition

Although one could conceive a definition in which each group in a normal series is a subgroup of the next, this creates an increasingly long chain of subtypes. If we have `H : subgroup G`, coercing `H` into a type results in the subtype `{x : G // x ∈ H}`. If we then have `K : subgroup H`, its coercion into a type will be `{x : ↥H // x ∈ K}`, where `↥H` is the subtype we just saw. We would instead like `↥K` to be `{x : G // x ∈ K}`, that is, a subtype of `G` and not of `H`. Because of the way the coercion of subgroups to types is set up, an element of `↥K` will have the form `⟨g, hH, hK⟩`, where `hH` and `hK` have types `g ∈ H` and `g ∈ K` respectively. But since `K` is a subgroup of `H`, `hH` follows from `hK`, which means that the former is unnecessary. The redundant information only increases in size when taking further subgroups of subgroups. This renders such a definition similarly unsatisfactory, and suggests that a change in perspective is in order.

Group theory deals with the consequences that distil from the group axioms. It is not concerned with what the elements of a group are, but with the ways they interact with each other. For this reason, a subgroup is seen as a group in its own right. What is important is not that its elements form a subset of a larger group, but that they multiply in a way that perfectly resembles how certain elements of the larger group multiply. When someone writes 'the cyclic group of order three is a subgroup of the group of symmetries of the triangle', they do not mean that the elements of the former are elements of the latter. Instead, they mean that the cyclic group of order three *embeds* into the group of symmetries of the triangle. An *embedding* is nothing more than an injective group homomorphism. Since it is injective, it is a bijection (and hence an isomorphism) onto its image. This way, instead of talking about subgroups of groups, we talk about groups embedding into other groups. In our case, we require an additional condition: the image of the embedding must be a normal subgroup.

```
structure normal_embedding (G H : Type*) [group G] [group H]
  extends φ : G →* H :=
(inj : function.injective φ)
(norm : φ.range.normal)
-- from src/jordanholder/normal_embedding.lean
```

Here `G →* H` denotes the type of (bundled) monoid homomorphisms, which are the standard way to represent group homomorphisms in *mathlib*. Thus, we have defined a normal embedding to be a monoid homomorphism together with the facts that it is injective and that its range[3] is normal. With this in mind, we can think of a normal series as a sequence of groups (no longer subgroups) with normal embeddings between them. This can be realised in Lean using an inductive type, not unlike a `list`, where each inductive step in the construction deals with terms of different types.

```
inductive normal_series : Π (G : Type u) [group G], Type (u+1)
| base {G : Type u} [group G] (hG : subsingleton G) : normal_series G
```

[3]The library uses the word 'range' as a synonym of 'image'.

```
3 | cons {H G : Type u} [group H] [group G]
4   (f : normal_embedding H G) (s : normal_series H) : normal_series G
```

There seems to be quite a bit to unpack in this definition, especially since we have to repeatedly request for instances of the `group` type class. It essentially falls under the nil-cons pattern. The requirement that a `base` normal series be constructed from a group that is a `subsingleton`[4] captures the fact that the only group that admits a normal series of length one (in the 'list of subgroups' fashion) is the trivial group.

One immediate simplification that can be made at this stage is switching to a bundled version of the `group` type class. This will unclutter the definition and save us writing `(G : Type u) [group G]` in the rest of our type signatures. In *mathlib*, this is achieved with the `Group` type, which belongs in the category theory part of the library. It is simply a structure that stores both a type and an instance of the `group` type class for it. The usual way to construct a `Group` from a type `G : Type*` is the function `Group.of`, that will attempt to derive a `group` instance for its argument. A term `G : Group` can be treated much in the same way as `(G : Type*) [group G]`, especially since `Group` has a coercion to sort that extracts the original type. This has the effect that `a : G` means the same thing for either definition of `G` in the last sentence. Another benefit of using the `Group` type is that it emphasises that we will be working across groups, as opposed to working with elements inside individual groups. The `Group` type can be parameterised with a universe variable with the notation `Group.{u}`.

```
1 inductive normal_series : Group.{u} → Type (u+1)
2 | base {G : Group} (hG : subsingleton G) : normal_series G
3 | cons (H G : Group)
4   (f : normal_embedding H G) (s : normal_series H) : normal_series G
5 -- from src/jordanholder/normal_series.lean
```

The resulting definition is a lot cleaner than the previous two, and it will serve as basis for us to develop the rest of the theory. Having defined what a normal series is, it is natural to continue defining what its factors are, and in turn define composition series. However, before we venture further into the world of definitions, let us do a sanity check on the one that we have just established.

When devising new definitions in a proof assistant, it is easy to trick oneself into thinking that they make sense without proving the first thing about them. It is always good practice to ensure that one can 'work' with a definition before moving on to defining more complex objects relying on it. In our case, there is one important result that requires only normal series and embeddings that we should be able to prove. Informally, if we have a normal series for a group G and know that G is isomorphic to some other group H, we can produce a normal series for H by considering every intermediate subgroup as a subgroup of H. Our definition makes this construction very easy to formalise, since we never considered the groups in the series to be subgroups. That way, if we have an isomorphism $\varphi : G \to H$, we can keep every term in a normal series for G and only change the last normal embedding to go into H instead of G. In terms of maps, this says that if we have a normal embedding $f : N \to G$, then the composite $\varphi \circ f$ is also a normal embedding. While in a textbook these

[4] The type class `subsingleton` is inhabited by all types α where any two elements are equal. Explicitly, `subsingleton α` gives access to a proof of `∀ (a b : α), a = b`.

would be phrased as lemmas, in Lean, embeddings are structures, so instead of lemmas we write definitions.

```
variables {G H N : Group}
def comp_mul_equiv (f : normal_embedding N G) (e : G ≃* H) :
  normal_embedding N H :=
{ φ := h.to_monoid_hom.comp f,
  inj := function.injective.comp h.left_inv.injective f.inj,
  norm := by rw range_comp; exact normal.mul_equiv_map f.norm h }
-- from src/jordanholder/normal_embedding.lean

def of_mul_equiv (e : G ≃* H) : normal_series G → normal_series H
| (base hG) := ...
| (cons K G f s) := cons K H (comp_mul_equiv f e) s
-- from src/jordanholder/normal_series.lean
```

The definitions are fairly natural. We used structure notation to provide a normal embedding in the first one, and gave a recursive definition for the second. I omitted the `base` case because it was pivotal in setting up the right definition. When I first wrote down the definition in terms of normal embeddings, it did not include the `subsingleton` assumption in the `base` case. The idea was to define a normal series that did not necessarily start with the trivial group. With such a definition, `of_mul_equiv` would simply return a new `base` normal series in the `base` case. This raised some suspicions: the assumption `e : G ≃* H` would be completely ignored. Moreover, we need at least two groups in a normal series to define its factors, and, for composition series, we will need a way to enforce that the starting group is trivial. This helped me realise that allowing normal series not starting with the trivial group would result in unnecessary complexity. In turn, this led to a definition in which the base case was fixed to be a normal series for the `punit` group (this is the unit type, which only contains one element). The new definition had its own problems: if we had a group `G` isomorphic to the trivial group `punit` then `of_mul_equiv` could no longer use the `base` constructor for a normal series of `G`. Eventually, this was solved by allowing any representation of the trivial group (that is, any group that is a `subsingleton`) to have a normal series built with the `base` constructor.

The final definition came with some small print, however: now there is *something* to do in the `base` case. Luckily, it is nothing we cannot fix with some already existing *mathlib* lemmas. After all, an isomorphism is a bijection, and if the domain is a subsingleton so must the codomain.

```
def of_mul_equiv (e : G ≃* H) : normal_series G → normal_series H
| (base hG) := trivial (@equiv.subsingleton.symm _ _ e.to_equiv hG)
| (cons K G f s) := cons K H (comp_mul_equiv f e) s
-- from src/jordanholder/normal_series.lean
```

As it stands, our definition has endured its first trial and is ready for further formalisation.

4.3 Developing the theory

The next steps are clear: we need to define the factors of a normal series and, using them, what it means to be a composition series. We will do so with our aim of proving the Jordan–Hölder theorem in mind. It is eventually a statement about the uniqueness of the factors of a composition series, only this uniqueness is up to permutation and isomorphism. Because of this, it would greatly simplify our task to define these factors in a way that already considered permutation and isomorphism irrelevant. Lists that are considered the same up to permutation are encoded in *mathlib* with the `multiset` type. It would be natural for us to define the factors of a normal series as a multiset of groups. This, however, does not take care of the fact that we will consider isomorphic groups as the same group. It turns out that *mathlib* also gives a way to overcome this obstacle: the `isomorphism_classes` functor[5] in the category theory section of the library.

Both of these types make use of quotients, which are one Lean's axiomatic extensions to its type theory. Given a type α and a relation `r : α → α → Prop` on α, the type `quot r` represents the quotient of α by the equivalence relation induced by `r`. Essentially, `multiset α` is the quotient of `list α` by the equivalence relation of whether two lists are permutations of each other. Similarly, `isomorphism_classes.obj (Cat.of Group)` is the quotient of the category of groups by the equivalence relation given by two groups being isomorphic. The ideal type for the factors of a normal series is then `multiset (isomorphism_classes.obj $ Cat.of Group)`. Then, saying that the factors of a composition series are the same up to permutation and isomorphism is to say that they are equal. This approach is entirely novel: the previous formalisations ([14], [21] and [20]) all defined factors as a list or sequence of groups. Each of these then goes on to define a predicate on lists of groups that expresses equivalence up to permutation and isomorphism. This is unnecessary in our case. Our definition is in fact only possible because Lean has quotient types, which, for instance, are not built into Coq. This is an example of how a more expressive type theory can enable simpler, arguably more natural definitions.

```
1 def factors : Π {G : Group}, normal_series G →
2   multiset (isomorphism_classes.obj $ Cat.of Group)
3 | _ (base _) := 0
4 | _ (cons H G f s) :=
5   quotient.mk' (Group.of $ quotient f.φ.range) ::ₘ factors s
6 -- from src/jordanholder/normal_series.lean
```

As could not be otherwise, we use a recursive definition. Note how we are performing recursion not only on the normal series, but also on the group, even though it appears as though we never use this argument. In reality, it is crucial that we do so, since the recursive call `factors s` is implicitly given `H : Group` as its first argument, and not `G`.

The two uses of quotient in the last line of the definition are not to be confused. The first (`quotient.mk'`) creates an isomorphism class from a given group, while the second constructs the quotient group from a normal subgroup, `f.φ.range` in this case. For the latter to have an associated group instance (we plan to make it into a term of type `Group` by applying `Group.of`), we need an instance of `f.φ.range.normal`. When we defined a nor-

[5] In fact, we only need the map between objects, which is accessed with `isomorphism_classes.obj`.

mal embedding, we included this normality condition as one of the fields of the structure. However, we have not exposed this to the type class resolution system. If we want Lean to be able to infer a group instance for `quotient f.φ.range`, we had better do so in advance.

```
instance (f : normal_embedding G H) : f.φ.range.normal := f.norm
-- from src/jordanholder/normal_embedding.lean
```

Once we have defined what the factors are, we are in position to tell Lean what a composition series is. In order to do this, we can simply define a subtype of normal series with the property that all of its factors are simple and nontrivial.

```
def composition_series (G : Group.{u}) : Type (u+1) :=
{ σ : normal_series G //
  ∀ G' ∈ σ.factors, is_simple_class G' ∧ ¬ is_trivial_class G' }
-- from src/jordanholder/normal_series.lean
```

Admittedly, we still need to define what it means for an isomorphism class to be simple and trivial. For this, we will use the `quot.lift` function, included in the quotient axiomatic extension. This is the type-theoretical equivalent of a mathematician saying that 'a function descends[6] to the quotient'. Given a function `f : α → β` and a proof that `f a = f b` whenever `a` and `b` are related by the quotient relation `r`, `quot.lift` defines a function `quot r → β`. Concretely, in our case we will have two predicates (`is_simple` and `is_trivial`) defining what it means for a group to be simple and trivial. Then, we can use `quot.lift` to 'lift' them into predicates on isomorphism classes of groups.

```
def is_simple (G : Type*) [group G] : Prop :=
∀ (N : subgroup G), N.normal → N = ⊥ ∨ N = ⊤

def is_simple_class (C : isomorphism_classes.obj (Cat.of Group)) : Prop :=
quotient.lift_on' C (λ (G : Group), is_simple G) ...
-- from src/jordanholder/simple_group.lean

def is_trivial_class (C : isomorphism_classes.obj (Cat.of Group)) : Prop :=
quotient.lift_on' C (λ (G : Group), subsingleton G) ...
-- from src/jordanholder/trivial_class.lean
```

We did not need to define a predicate `is_trivial`, since *mathlib* already comes with `subsingleton`, which is equivalent in the case of groups. Note that here, as before, we did not explicitly use the `quot` namespace, nor did we use the `lift` function. Instead, we used *mathlib*'s `quotient` namespace. The difference between the two is that the primitive `quot` is defined for any relation, while `quotient` needs an equivalence relation. The type of isomorphism classes is defined internally using `quotient`, so we need to use the functions in the `quotient` namespace. The reason for `lift_on'` instead of `lift` is more technical. The function `quotient.lift` takes the necessary equivalence relation (`setoid` in *mathlib*-speak) as a type class argument. However, the equivalence relation at hand (whether two groups are isomorphic) is not declared as an instance of the `setoid` type

[6] One can tell whether somebody is a computer scientist or a mathematician depending on how they call this action. For a computer scientist, the quotient type is larger (higher), in the sense that it encodes more information. For a mathematician, a quotient is smaller (lower) than the original set. Hence, the former speak of 'lifting a function', whereas the latter speak of 'a function descending'.

class. There are two ways of addressing this: declaring the corresponding instance or using type inference, instead of type class resolution, to provide the necessary `setoid`. The function `quotient.lift_on'` does precisely the latter. For comparison, here are the type signatures of `quotient.lift_on` and `quotient.lift_on'`.

```
quotient.lift_on {α : Sort u} {β : Sort v} : Π [s : setoid α], quotient s →
  Π (f : α → β), (∀ (a b : α), a ≈ b → f a = f b) → β

quotient.lift_on' {α : Sort u} {β : Sort v} : Π {s : setoid α}, quotient s →
  Π (f : α → β), (∀ (a b : α), @setoid.r α s a b → f a = f b) → β
```

The benefits of deriving our `setoid` using type class resolution is that it allows us to use notation such as `a ≈ b` for the equivalence relation, or `⟦a⟧` for the equivalence class of `a`. However, this option is usually reserved for when there is only one sensible instance for a given type, and it is not clear whether this is the case with our `setoid`. In any case, it is preferable to use type inference when it is available, so we will use the `quotient.lift_on'` version. The theory could have very well been developed using the other approach, and perhaps more succinctly in some parts, but we will not do so here.

Lastly, note that in our definition of `is_simple_class` and `is_trivial_class` we did not simply give `is_simple` and `subsingleton` as the predicates to be lifted. Instead, we gave a lambda expression which calls each predicate on the given argument. To the naked eye, this seems completely unnecessary, but once again it is no accident. The predicates `is_simple` and `subsingleton` do not take a `Group` as their argument, instead they take a `Type*`. Therefore, when we wrote `subsingleton G`, there is an implicit coercion of `G` from `Group` to `Type*`. In the case of `simple`, not only is there a coercion to `Type*`, but also there is a `group` instance derived by the elaborator.

Before taking on the proof of the existence of composition series and the Jordan–Hölder theorem, it will be useful to become acquainted with our definition by proving some useful facts. In particular, note that the defining property of a composition series is very closely related to the underlying normal series. We could have defined composition series as a new inductive type, much like we did for normal series, with the added assumption at each stage that the quotient associated to each normal embedding is simple and nontrivial. Instead, we used a more modular definition, that will allow us to prove lemmas about normal series and then use them in the context of composition series. Nevertheless, the inductive type view is useful when one wants to construct a composition series from an old one. Instead of making it the definition, we will define two functions analogous to what the inductive type constructors would be. The analogue of `base` is `of_subsingleton` and the analogue of `cons` is `cons'`. Together, these two lemmas provide an interface similar to the one we would expect if we had defined composition series from scratch as an inductive type.

```
def of_subsingleton (h : subsingleton G) : composition_series G :=
⟨base h, λ G' hG', absurd hG' $ multiset.not_mem_zero _⟩

def cons' (σ : composition_series H) (f : normal_embedding H G)
(h : is_simple (quotient f.φ.range) ∧ ¬subsingleton (quotient f.φ.range)) :
  composition_series G :=
⟨cons _ _ f σ.val, λ G' hG',
```

```
8   by { simp at hG', cases hG', { simpa [hG'] }, exact σ.prop _ hG' })
9 -- from src/jordanholder/normal_series.lean
```

4.3.1 Existence of a composition series

With our definitions set, we are ready to start working on our first goal: proving that every finite group has a composition series. The standard way to do this is via maximal normal subgroups. A maximal normal subgroup of a group G is a proper normal subgroup N that is not properly contained in any other normal subgroups, except for G itself. It turns out that N is a maximal normal subgroup of G if and only if G/N is simple and nontrivial. Therefore, we can use the normal embedding $N \hookrightarrow G$ as our last step in a composition series for G. Because every finite group has a maximal normal subgroup, we can repeat this process inductively to define a composition series for it.

We define maximal normal subgroups in Lean as a predicate on subgroups. Note that we have to include an instance for `[N.normal]` in the lemma, since otherwise the quotients would not be groups, and `is_simple` is a predicate about groups.

```
1 def maximal_normal_subgroup (N : subgroup G) : Prop :=
2   N.normal ∧ N ≠ ⊤ ∧ ∀ (K : subgroup G) [K.normal], N ≤ K → K = N ∨ K = ⊤
3 -- from src/jordanholder/subgroup.lean
4
5 lemma maximal_normal_subgroup_iff (N : subgroup G) [N.normal] :
6   maximal_normal_subgroup N ↔
7   is_simple (quotient N) ∧ ¬subsingleton (quotient N) :=
8 ⟨λ hN, ⟨begin
9   intro K, introI,
10  have : N ≤ comap (mk' N) K := ...,
11  refine (hN.2.2 (comap (mk' N) K) this).imp ...,
12 end,
13   λ h, hN.2.1 (subsingleton_quotient_iff.mp h)⟩,
14 λ ⟨h1, h2⟩, ⟨infer_instance, λ h, h2 (subsingleton_quotient_iff.mpr h), begin
15     intro K, introI, intro hNK,
16     refine (h1 (map (mk' N) K) (map_mk'_normal hNK)).imp ...,
17 end⟩⟩
18 -- from src/jordanholder/fingroup.lean
```

The lemma characterising maximal normal subgroups makes implicit use of the subgroup correspondence theorem, which states that the quotient map $p : G \to G/N$ induces a bijection between the subgroups of G containing N and the subgroups of G/N. The crucial property is that this bijection preserves inclusions and normality. When we prove the `is_simple` part in the forward implication, we are challenged with a normal subgroup K of G/N and our task is to prove that it is either the trivial subgroup or the entire group. In Lean, the quotient map p is written `mk' N`, and the preimage of `K` under it is `comap (mk' N) K`. In line 11, we use the fact that `comap (mk' N) K` is a normal subgroup of `G` containing `N`, which is a maximal normal subgroup, so `K` must be either `N` or `⊤`. Note that we do not explicitly prove that `comap (mk' N) K` is normal in `G`. Instead this is handled by type class resolution, since we wrote `[K.normal]` in square brackets in line 2. The library already has a `normal` instance for the preimage of any normal subgroup under

a group homomorphism, so there is no work for us here.

When proving maximality of `N` from the assumption that `quotient N` is simple, we are again given a challenge normal subgroup `K` of `G` containing `N` and have to prove that it is either `N` or `⊤`. In this case, we do have to provide a proof that `map (mk' N) K` (the analogue of $p(K)$) is normal, which is given by the `map_mk'_normal hNK` term in line 16. This happens for two reasons. Firstly, when we defined `is_simple` the normality assumption was not inside square brackets, so type class resolution is not automatically invoked. Secondly, it is not true that the image of a normal subgroup under a group homomorphism is normal, so there is no general instance in the library in this case. This is valid, however, when the homomorphism is the quotient map and under the assumption that $N \leq K$ (here `hNK`).

Intuitively, it is clear that any nontrivial finite group has a maximal normal subgroup. To find one, start with the trivial subgroup (which is always normal). If it is not maximal, then there is some proper normal subgroup that strictly contains it. If this second normal subgroup is not maximal, we repeat the process. We are guaranteed to end at some point because there are only finitely many subgroups. Unfortunately, this simple argument cannot be easily written in Lean. Instead, we will show that every nontrivial finite group G has a maximal normal subgroup by induction on the order of G.

In order to do this, we will need an induction principle, or more generally a recursion principle, on finite groups. We will provide two versions of this: one for the bundled type `Group`, and one for types members of the `group` type class. We will use the latter in our proof about maximal normal subgroups, since it is more closely related to the existing group-theoretic results in *mathlib*, all of which use the unbundled `group` type class. The former will be used in the proofs of the theorems about normal and composition series, since those are defined for elements of type `Group`. In either case, the statement we are trying to formalise is: given a target function type from finite groups to some other type (if the target type is `Prop`, this function is a predicate) and, for any finite group G, a way to define this function at G given its definition at any group H with strictly smaller order, we can define the target function on all finite groups. We will only show the formalised version in the case of bundled groups, the other is completely analogous.

```
1  def strong_rec_on_card' (G : Group) [fintype G]
2    {p : Π (G : Group) [fintype G], Sort _}
3    (ih : Π (G : Group) [fintype G],
4      (Π (H : Group) [fintype H], by exactI card H < card G → p H) →
5      by exactI p G) :
6    p G :=
7  suffices h : Π (n : ℕ) (G : Group) [fintype G],
8    by exactI card G = n → p G,
9    from h (card G) G rfl,
10 λ n, n.strong_rec_on ...
11 -- from src/jordanholder/fingroup.lean
```

This definition manifests one of the limitations of the type class system. For performance reasons, only the instances declared at the top level in a type signature are added to the instance cache. For example, when we introduce `[fintype G]` as the second argument to `strong_rec_on_card'`, `fintype G` is stored automatically, and it is retrieved when

needed. This way, `p G` in line 6 can find the instance required by `p` in the cache. Similarly, `card G` in line 9 silently uses the stored `fintype G` instance. By contrast, in line 4 we define a function that takes `H : Group`, derives an instance `fintype H`, and takes a proof of `card H < card G` to produce a term of type `p H`. Because `[fintype H]` is not in the top level of the overall definition, Lean does not add it to the instance cache, not even in the context where `H` is defined. How are we then allowed to write `card H < card G` and `p G` if a `fintype H` instance is not available? The answer is by using the `exactI` tactic. This tactic takes some expression (in this case `card H < card G → p H`) and uses it to close the goal (here a return type for the function we are defining), while using all available variables for type class inference (here `G`, `fintype G`, `H` and `fintype H`). Essentially, `exactI` resets the instance cache temporarily to include everything in the current context. This allows us to manually use instances that are not declared in the top level of our definition. We use the same tactic in lines 5 and 8 to ensure that `fintype G` is available.

The `exactI` tactic is part of a family of tactics (most of which end in a capital 'I') that allow us to manipulate the instance cache. Another important member of this family is `introI`. When the goal is of the form `Π [fintype G], _` using lambda abstraction or the `intro` tactic will give a `fintype G` instance, but not add it to the cache. If we want it to be used for type class inference in the future, we should introduce it with the `introI` tactic instead. When using it, it is common to leave the instances unnamed – we just write `introI`. This is because we are not planning to use them explicitly, since the type class resolution system will handle them for us.

As for the definition itself, it first introduces an argument `n : ℕ` that will act as the order of the group. This transforms the goal into a function of natural numbers, that can be defined using the strong recursion principle in ℕ. Since `Prop` is just `Sort 0`, we can use `strong_rec_on_card'` as an induction principle. In fact, this will be our first use for it – or rather for `strong_rec_on_card`, the version for unbundled groups.

```
lemma exists_maximal_normal_subgroup [fintype G] :
  ¬ subsingleton G → ∃ (N : subgroup G), maximal_normal_subgroup N :=
strong_rec_on_card G begin
  intro G, introsI _ _, intros ih hG,
  by_cases h : is_simple G,
  { use [⊥], ... },
  rcases not_is_simple.mp h with ⟨N, hN, hN'⟩, haveI := hN,
  rcases ih (quotient N) (card_quotient_lt hN'.1) ...
    with ⟨K, hK, hKtop, hKmax⟩,
  use [comap (mk' N) K], ...,
end
-- from src/jordanholder/fingroup.lean
```

The proof splits into two cases depending on whether G is simple (line 5). If it is, then the trivial subgroup (⊥) is a maximal normal subgroup (line 6). Otherwise, we can find a nontrivial proper normal subgroup N (line 7). We can then apply the inductive hypothesis to G/N to find a maximal normal subgroup K (lines 8 and 9). If we write $p : G \to G/N$ for the quotient map, then $p^{-1}(K)$ – the familiar `comap (mk' N) K` – is a maximal normal subgroup of G (line 10).

We are now close to completing the proof that every finite group has a composition series. The last step involves formalising the idea that, if N is a maximal normal subgroup of G, then we can use the embedding $N \hookrightarrow G$ as the last in our composition series.

```
def cons_of_maximal_normal_subgroup {N : subgroup G}
  (h : maximal_normal_subgroup N) (σ : composition_series (Group.of N)) :
  composition_series G :=
begin
  haveI := h.1, apply cons' σ (of_normal_subgroup N),
  have h' := (maximal_normal_subgroup_iff N).mp h, split,
  { rw is_simple_quotient_eq (range_of_normal_subgroup N), exact h'.1 },
  { rw range_of_normal_subgroup, exact h'.2 }
end
-- from src/jordanholder/normal_series.lean
```

First, we introduce the `N.normal` instance provided into the instance cache using the `haveI` tactic. We then use the previously defined `cons'` with the provided composition series and the result of `of_normal_subgroup N`, which is the natural normal embedding of a normal subgroup. It then remains to prove that the associated quotient is simple and nontrivial. This is handled by lines 7 and 8. Even though this is a definition, we can write it in tactic mode. Most tactics are not specialised for terms of type `Prop`, and they can be used when the target is a type containing data.

We have everything we need to prove our main result for this section.

```
theorem exists_composition_series_of_finite [fintype G] :
  nonempty (composition_series G) :=
strong_rec_on_card' G begin
  intro G, introI, intro ih,
  by_cases h : subsingleton G,
  { use of_subsingleton h },
  rcases exists_maximal_normal_subgroup h with ⟨N, hN⟩, haveI := hN.1,
  apply (ih (Group.of N) (card_lt hN.2.1)).elim,
  intro σ, use cons_of_maximal_normal_subgroup hN σ,
end
-- from src/jordanholder/normal_series.lean
```

As expected, we use the recursion principle – this time the version for the bundled `Group` type. There are two cases. Firstly, if G is the trivial group, then we have already defined a composition series for it, namely `of_subsingleton`. Otherwise, G has a maximal normal subgroup N, which by induction has a composition series. It only remains to use `cons_of_maximal_normal_subgroup` to construct a composition series for G.

The use of `nonempty` the return type of this theorem is worth discussing. Even though the name of the theorem explicitly mentions the word 'exists', we do not use the existential quantifier. We are not proving that there exists a composition series 'such that ...', only that there exists one. This is achieved with `nonempty`, which is a predicate on some type (in this case `composition_series G`) saying that it contains at least one element. Lean's core also contains a similar type class called `inhabited`, which applies to types that have at least on element too. There is an important distinction between the two: if we have `α : Type u`,

then `nonempty α` has type `Prop`, while `inhabited α` has type `Type u`. This is because `inhabited α` actually stores an element of type α, which is retrieved with `default α` and which can be used in computable definition. On the other hand, `nonempty α` is merely a statement of the fact that there is some element in α, which can be achieved because `Prop` is impredicative. In our case, `nonempty` is the right choice because we are not explicitly constructing a member of `composition_series G`, as in the process we use the fact that there exists a maximal normal subgroup, which is noncomputable.

4.3.2 The second isomorphism theorem

Before we proceed with the proof of the Jordan–Hölder theorem, there is one key ingredient that we need that is missing from *mathlib*: the second isomorphism theorem. It states a number of facts about how subgroups and normal subgroups can be combined within a group, and proves that two of the resulting quotient groups are isomorphic. Suppose we have a group G and two subgroups H and N. We write HN for the subset $\{hn : h \in H, n \in N\}$. The second isomorphism theorem states that, if N is normal, then (i) HN is a subgroup of G, (ii) N is normal in HN, (iii) $H \cap N$ is normal in H, and (iv) $HN/N \cong H/(H \cap N)$.

First of all, we will need to define what the product of two subgroups HN is, and prove that it is a subgroup. Luckily, *mathlib* already comes with the pointwise set product operation, which overloads the `*` operator. Moreover, the group theory section of the library includes a proof (an instance) that the type of subgroups of a group forms a complete lattice. This allows as to talk about the least upper bound of a pair of subgroups H and N, which is the smallest subgroup that contains both H and N. In mathematics, this is usually called the join of H and N, and is written $H \vee N$. Similarly, there is the notion of the greatest lower bound of H and N, which is the largest subgroup contained in both H and N. In lattice terminology, this is called the join of H and N, written $H \wedge N$. In lean, join is written `H ⊔ N` and meet, `H ⊓ N`.

Because $H \cap N$ is itself a subgroup, we have $H \wedge N = H \cap N$. As a set, HN contains both H and N. However, it need not be a subgroup, and so in general $H \vee N \neq HN$. Equality holds if at least one of H and N is normal. Given this, it is natural to formalise the fact that HN is a subgroup when N is normal as a lemma stating that $H \vee N = HN$. We do this in three steps.

```
lemma sup_eq_closure (H K : subgroup G) : H ⊔ K = closure (H * K) := ...

private def mul_normal_aux (H N : subgroup G) [N.normal] : subgroup G :=
{ carrier := H * N, ... }

lemma mul_normal (H N : subgroup G) [N.normal] : (↑(H ⊔ N) : set G) = H * N :=
set.subset.antisymm
  (show H ⊔ N ≤ mul_normal_aux H N,
    by { rw sup_eq_closure, apply Inf_le _, dsimp, refl })
  ((sup_eq_closure H N).symm ▸ subset_closure)
-- from src/to_mathlib/pointwise.lean
```

First, we prove that, for subgroups `H` and `K` of `G`, `H ⊔ K` is the smallest subgroup containing

`H * K`. Then, given that `N` is normal, we define a subgroup of `G` whose underlying set is `H * N`. We make this a private definition because we never intend to use it explicitly; instead, it does half of the work needed to prove `mul_normal`. Using these two intermediate results, we can show that `H ⊔ K` is contained in `H * N` (lines 8 and 9) and vice versa (line 10). Now, the second isomorphism theorem assumes that `N` is normal, so we can use `H ⊔ K` in place of `H * K` resting assured that they are the same set.

We have just proved item (i) above. For the remaining ones, we will need a way of transferring terms of type `subgroup G` to `subgroup H`, where `H : subgroup G`. For any such `H`, `H.subtype` is the inclusion group homomorphism from `H` (coerced into a type) to `G`. Then if `K` is some other subgroup of `G`, the term `comap H.subtype K` will be a subgroup of `H`. Moreover, if `K ≤ H`, this contains 'essentially' the same elements as `K`. Formally, terms of type `H` have the form `⟨x : G, hx : x ∈ H⟩`, and `⟨x, hx⟩ ∈ comap H.subtype K` if and only if `x ∈ K`. More generally, even when `K` is not contained in `H` we have the following.

```
lemma comap_subtype_inf_left {H K : subgroup G} :
  comap H.subtype (H ⊓ K) = comap H.subtype K :=
ext $ λ x, and_iff_right_of_imp (λ _, x.prop)
-- from src/to_mathlib/sndiso.lean
```

Here, `ext` allows us to prove that two subgroups are the same by proving that an element `x` is in one of them if and only if it is in the other. We are now ready to prove parts (ii) and (iii) of the theorem. In fact, there is nothing to do for part (ii). As we saw before, *mathlib* already has an normal instance for the preimage of any normal subgroup. Since saying that N is normal in HN amounts to saying that `comap (H ⊔ N).subtype N` is normal, this instance covers case (ii). Not much work is required for part (iii) either.

```
instance subgroup.normal_inf (H N : subgroup G) [hN : N.normal] :
  (comap H.subtype (H ⊓ N)).normal :=
by { rw comap_subtype_inf_left, apply_instance }
-- from src/to_mathlib/sndiso.lean
```

The lemma we just proved, `comap_subtype_inf_left`, rewrites our subgroup as the preimage of `N` under `H.subtype`, and then the same instance as before finishes the job. Arguably, we could have done without this second instance if we wrote our theorem in terms of `comap H.subtype N` instead of `comap H.subtype (H ⊓ N)` – after all, they are the same subgroup. The second notation, however, would be far more familiar to a mathematician trying to find the second isomorphism theorem in the library.

There is one last element that we will need in the proof of part (iv), and which should probably be part of any library formalising group theory. We define the inclusion homomorphism from a subgroup `H` contained in `K` to `K` (this time, both subgroups must be coerced into types).

```
def inclusion {H K : subgroup G} (h : H ≤ K) : H →* K :=
monoid_hom.mk' (λ ⟨x, hx⟩, ⟨x, h hx⟩) (λ ⟨a, ha⟩ ⟨b, hb⟩, rfl)
-- from src/to_mathlib/sndiso.lean
```

The function `monoid_hom.mk'` allows us to construct a monoid homomorphism (remember that group homomorphisms in Lean are just monoid homomorphisms) from a function and a proof that it preserves multiplication. Our function is `λ ⟨x, hx⟩, ⟨x, h hx⟩`, which

given an element `x : G` and the fact that `x ∈ H` returns the same element with a proof that `x ∈ K`. The fact that it preserves multiplication is trivial, since multiplication happens inside the group `G` both in `H` and `K`. We are now ready to prove the last part of the second isomorphism theorem.

```
noncomputable def quotient_inf_equiv_prod_normal_quotient
(H N : subgroup G) [N.normal] :
  quotient (comap H.subtype (H ⊓ N)) ≃* quotient (comap (H ⊔ N).subtype N) :=
let φ : H →* quotient (comap (H ⊔ N).subtype N) :=
  (mk' (comap (H ⊔ N).subtype N)).comp (inclusion le_sup_left) in
have φ_surjective : function.surjective φ := λ x, x.induction_on' $
  begin
    rintro ⟨y, (hy : y ∈ ↑(H ⊔ N))⟩, rw mul_normal H N at hy,
    rcases hy with ⟨h, n, hh, hn, rfl⟩,
    use [h, hh], apply quotient.eq.mpr, change h⁻¹ * (h * n) ∈ N,
    rwa [←mul_assoc, inv_mul_self, one_mul],
  end,
(equiv_quotient_of_eq (by simp [comap_comap, ←comap_ker])).trans
  (quotient_ker_equiv_of_surjective φ φ_surjective)
-- from src/to_mathlib/sndiso.lean
```

We begin by defining an auxiliary homomorphism, φ, as the composition of the quotient map onto `quotient (comap (H ⊔ N).subtype N)` and the inclusion of `H` in `H ⊔ N`. Next, we prove that φ is surjective. We are given some element `x` of the quotient, and are challenged to find some element of `H` that maps to it. A standard way to work with terms of a quotient type is their induction principle (understood more as a recursor than, say, induction of natural numbers). Essentially, if we want to prove a fact about an equivalence class `x`, it is enough to prove the corresponding fact about every element `y` in it. Given any `y ∈ H ⊔ N` that maps to `x` under the quotient map, the fact that `H ⊔ N` as a set is just `H * N` (line 8) gives us `h ∈ H` and `n ∈ N` such that `y = h * n` (line 9). It then follows that `φ h` is the coset of `N` that contains `y` (lines 10 and 11). The last two lines complete the proof by using the first isomorphism theorem (written `quotient_ker_equiv_of_surjective`), thus showing that the quotient of `H`, which is exactly `comap H.subtype (H ⊓ N)`, is isomorphic to the codomain of φ. Because the first isomorphism theorem is not computable, neither is the second.

Once one becomes familiar with the `comap H.subtype` notation, this proofs reads very much like the one you might find in a group theory textbook – provided, of course, that you use Lean's interactive view to see the state of the goal at each step. The only point that would never be mentioned in an informal proof is the use of `equiv_quotient_of_eq` in line 13. The `simp` tactic gives a proof that `comap H.subtype (H ⊓ N) = φ.ker`, which is crucial part of the proof. Using the first isomorphism theorem gives us and isomorphism `quotient φ.ker ≃* quotient (comap (H ⊔ N).subtype N)`. Rewriting the previous result on the left hand side of this isomorphism should finish the proof. However, this substitution can only be performed when the resulting goal is of type `Prop`, and an isomorphism in Lean (in this case a `mul_equiv`, written `≃*`) is in fact an object, not a proposition. That is because it stores two inverse functions, together with a proof that they preserve multiplication. Instead of rewriting, we use `equiv_quotient_of_eq` which has the following type.

```
def equiv_quotient_of_eq {M N : subgroup G} [M.normal] [N.normal] (h : M = N) :
  quotient M ≃* quotient N :=
by unfreezingI { subst h }
-- from src/to_mathlib/sndiso.lean
```

It may be hard to convince oneself that this definition is doing anything at all. First, we use the `unfreezingI` tactic, which executes the tactics in the block following it allowing the instance cache to be modified freely. The reason for this is that even though rewriting would give us `quotient M = quotient N`, an isomorphism is also a statement about the multiplicative structures. Since these are hidden in the form of instances, we will need to examine the contents of the cache to show that these instances (and not just the underlying types) are equal. The work of showing these equalities is done by the `subst` tactic, which uses the given assumption `h : M = N` everywhere it needs to in order to close the goal. This conveniently hides heterogeneous equality[7] issues, of which we can luckily remain oblivious. With this result, it is just a matter of using the transitivity of isomorphism to finish our proof. Informally, if N and M are the same normal subgroup of G, then without a doubt one can say that G/N and G/M are the same group. But, as we have seen, it takes a bit of care to formalise this.

4.3.3 Uniqueness: the Jordan–Hölder theorem

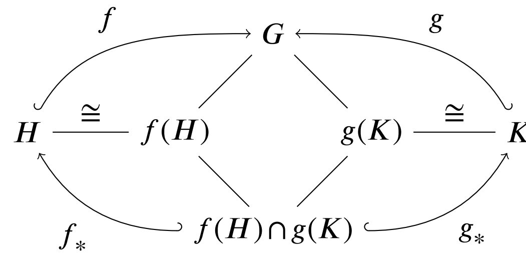
Most requirements are now set up for us to prove the Jordan–Hölder theorem. Almost all of what remains to be done before the final result can be seen as paying the toll of using normal embeddings instead of subgroups. For instance, we have proven the second isomorphism theorem in its full generality – it applies to any type that has an instance of the `group` type class, including the terms of type `Group`. Since composition series do not deal with normal subgroups, but with normal embeddings, the theorem would be easier to use if it was phrased in terms of the latter. In fact, there is one particular scenario in which the second isomorphism theorem applies that is instrumental to our proof of the Jordan–Hölder theorem. We formalise it as follows.

```
noncomputable def equiv_range (f : normal_embedding G H) : G ≃* f.φ.range :=
mul_equiv.of_injective f.inj

noncomputable def from_inf_range_left
(f : normal_embedding H G) (g : normal_embedding K G) :
  normal_embedding ↥(f.φ.range ⊓ g.φ.range) H :=
comp_mul_equiv (of_normal_subgroup_to_subgroup inf_le_left) (equiv_range f).symm

noncomputable def quotient_from_inf_range_left
(f : normal_embedding H G) (g : normal_embedding K G)
(h : f.φ.range ⊔ g.φ.range = ⊤) :
  quotient (from_inf_range_left f g).φ.range ≃* quotient g.φ.range := ...
-- from src/jordanholder/normal_embedding.lean
```

First, we state the fact that if `f` is a normal embedding from `G` to `H`, then `G` is isomorphic

[7] Heterogeneous equality is used to express equalities of terms whose types are not definitionally equal. Lean uses the `==` operator for this. In our case, we have terms `g1 : group (quotient M)` and `g2 : group (quotient N)`. Writing `g1 = g2` does not typecheck, but `g1 == g2` does.

to the image of `f`. Next, we transfer part (iii) of the second isomorphism theorem to the language of normal embeddings. Perhaps a diagram is in order.



Given normal embeddings `f` and `g`, `from_inf_range_left f g` gives the normal embedding labelled f_*, that is, the composition of the normal embedding from $f(H) \cap g(K)$ as a normal subgroup of $f(H)$ and the inverse of the isomorphism $H \cong f(H)$. Lastly, the definition of `quotient_from_inf_range_left` transfers a special case of part (iv) of the theorem, as it relates the quotient of $f(H) \cap g(K)$ in H to the quotient of K in G. It is less general, because of assumption `h` in its type signature. This assumption simplifies the diagram, as we only need one vertex for $f(H) \vee g(H) = G$. Along with the last two definitions, I also wrote down `from_inf_range_right` and `quotient_from_inf_range_right`, which give g_* in the diagram, with the corresponding isomorphism. These are all noncomputable because they all depend on `mul_equiv.of_injective` on line 2. It could not be otherwise, since in the definition of a normal embedding we only specified that it had to be injective; we did not require that it have computable left inverse.

The following lemmas may shed some light on why we only need the isomorphisms in the previous paragraphs under the assumption that $f(H) \vee g(H) = G$.

```
1  lemma sup_maximal_normal_subgroup {N K : subgroup G}
2    (hN : maximal_normal_subgroup N) (hK : maximal_normal_subgroup K)
3    (h : N ≠ K) : N ⊔ K = ⊤ := ...
4  -- from src/jordanholder/subgroup.lean
5
6  lemma maximal_normal_subgroup_of_cons {σ : composition_series G}
7    {f : normal_embedding H G} {s : normal_series H} :
8    σ.val = cons H G f s → maximal_normal_subgroup f.φ.range := ...
9  -- from src/jordanholder/normal_series.lean
```

The first says that the join of any two distinct maximal normal subgroups N and K is the entire group. This follows easily from the maximality condition and the fact that $N \vee K$ is a normal subgroup containing both N and K. The second one states that the image of any normal embedding used in a composition series is a maximal normal subgroup. It is a direct consequence of our earlier characterisation of maximal normal subgroups as those whose quotient is simple and nontrivial.

We are now ready to prove the Jordan–Hölder theorem for finite groups. Note how the statement of the theorem greatly capitalises on our chosen definition of factors. If our factors had been, say, of type `list Group`, we would then have to formalise what it means for two lists to be the same up to reordering and isomorphism. Since these requirements are already encoded in the type of factors, our statement just asserts that the factors of any two composition series for `G` are the equal.

```
theorem eq_factors [fintype G] : Π (σ τ : composition_series G),
  σ.val.factors = τ.val.factors :=
strong_rec_on_card' G begin
  intros G, introI, intro ih, intros σ τ,
  by_cases hG' : subsingleton G, { simp only [factors_of_subsingleton hG'] },
  rcases exists_cons_of_not_subsingleton hG' σ.val with ⟨H, f, s, hs⟩,
  rcases exists_cons_of_not_subsingleton hG' τ.val with ⟨K, g, t, ht⟩,
  simp [hs, ht],
  by_cases f.φ.range = g.φ.range,
  { congr' 1, { exact class_eq (equiv_quotient_of_eq h) }, ... },
  have htop := sup_maximal_normal_subgroup (maximal_normal_subgroup_of_cons hs)
    (maximal_normal_subgroup_of_cons ht) h,
  apply (exists_composition_series_of_finite
    (Group.of ↥(f.φ.range ⊓ g.φ.range))).elim, intro ρ,
  let s' : composition_series H := cons' ρ (from_inf_range_left f g) ...,
  let t' : composition_series K := cons' ρ (from_inf_range_right f g) ...,
  have hs' := @ih H f.fintype (card_lt_of_cons hs) s' (of_cons hs),
  have ht' := @ih K g.fintype (card_lt_of_cons ht) t' (of_cons ht),
  rw [factors_of_cons hs, factors_of_cons'] at hs', rw ←hs',
  rw [factors_of_cons ht, factors_of_cons'] at ht', rw ←ht',
  conv_rhs { rw multiset.cons_swap }, congr' 1,
  { exact class_eq (quotient_from_inf_range_right f g htop).symm },
  { congr' 1, exact class_eq (quotient_from_inf_range_left f g htop) },
end
-- from src/jordanholder/normal_series.lean
```

As promised, we proceed by induction on the order of `G`. We first take care of the case where `G` is the trivial group (line 5). Otherwise, `σ` and `τ` must be built with the `cons` constructor, so we can retrieve groups `H` and `K`, normal embeddings `f` and `g` and series `s` and `t` respectively. Next, we consider two cases. If `f.φ.range = g.φ.range`, then the last step has the same factor (line 10), and `s` and `t` must too by the induction hypothesis. The other case begins by showing that `f.φ.range ⊔ g.φ.range = ⊤`, using `sup_maximal_normal_subgroup` (lines 11 and 12). Then we use the existence theorem to procure a composition series for `f.φ.range ⊓ g.φ.range` (lines 13 and 14). We use this to build composition series `s'` and `t'` for `H` and `K` respectively, which must have the same factors as `s` and `t` by induction (lines 15–18). This shows that `s` and `t` have almost the same factors, except for one: the last steps in `s'` and `t'`. We are, however, interested in the factors of `σ.val = cons H G f s` and `τ.val = cons K G g t`. The second isomorphism theorem then shows that the factor that `f` introduces is the factor that `s` was missing, and `g` introduces the factor that `t` was missing (lines 22 and 23). Since we are dealing with multisets, the order in which these are added is irrelevant (line 21), and so the factors of `σ` and `τ` are the same.

The `congr` tactic used repeatedly in this proof attempt to use a series of congruence lemmas. These are related results that say that if two functions are equal and they are applied to terms that are equal, the results are also equal. For instance, at the beginning of line 10, the goal state is as follows.

```
⊢ quotient.mk' (Group.of (quotient f.φ.range)) ::ₘ s.factors =
```

```
  quotient.mk' (Group.of (quotient g.φ.range)) ::ₘ t.factors
```

If we applied `congr` unrestricted, it would reduce it to two goals: showing `f == g` and `s = t`. Of course, proving that would close the current goal, but it is simply not true. This is an example of overly aggressive automated simplification. Instead we specify that `congr` should only simplify the goal once. This is done with `congr' 1`. After this, the goal is split into two subgoals, which are provable.

```
⊢ quotient.mk' (Group.of (quotient f.φ.range)) =
  quotient.mk' (Group.of (quotient g.φ.range))
⊢ s.factors = t.factors
```

This theorem finishes our development of composition series in Lean. In the next section, we will use these results to prove a toy corollary.

4.4 The fundamental theorem of arithmetic

Formalising a new theory is all well and good, but how can we guarantee that it is useful in its current form? I made numerous choices along the way. Sometimes, these were intended to align better with the library, such as when we used embeddings instead of subgroups. Others, however, their purpose was to facilitate the job of proving certain results, particularly when we defined factors as a multiset of isomorphism classes. There is no doubt that the statement of the Jordan–Hölder theorem given is equivalent to the informal one, but perhaps it has been formalised in way that makes its use in further results cumbersome. We address this in this section.

The best way to demonstrate the usefulness of our results is to use them to prove further results. A standard corollary of the Jordan–Hölder theorem, as shown in [18], is the fundamental theorem of arithmetic. In particular, the statement that any two prime factorisations of a positive natural number n are equal up to reordering. It follows from applying Jordan–Hölder to the additive cyclic group $\mathbb{Z}/n\mathbb{Z}$ of integers modulo n. We will see that any factorisation of n gives rise to a composition series of $\mathbb{Z}/n\mathbb{Z}$, whose factors' orders are the prime numbers in the factorisation.

Even if the corollary can seem fairly direct when proved informally, there is some work to do in Lean before we can show the implication. It begins with transferring all of our results in the previous section to additive notation. In *mathlib*, there are two different type classes for groups: `group` and `add_group`. Since the `*` notation is borrowed from the group instance, we need a separate type class for groups written in additive notation. Luckily, the library comes with a custom attribute (`to_additive`) that makes our job close to trivial. It uses Lean's metaprogramming capabilities to automatically transfer lemmas and definitions from multiplicative notation to additive notation. For most of the results in the previous sections, adding `@[to_additive]` on the line above creates a copy of the result in the corresponding additive namespace. If there is no analogous namespace, one can provide a new name for the additive translation. The one limitation of this attribute is that it cannot automatically translate inductive type and structure definitions. These must be rewritten manually, and only then is `to_additive` be able to associate the two versions. Except for these cases, the `to_additive` attribute does a great job of hiding the fact that

multiplicative and additive groups are completely separate type classes. This is good news for any mathematician planning to formalise group theory results in *mathlib*.

Next, we need to define the cardinality of an isomorphism class. This, in turn, needs defining what a finite class is. For finiteness, we can simply lift the predicate of a group having some `fintype` instance. The case of cardinality is not so simple. We cannot lift the `card` function to the quotient type, because it only applies to isomorphism classes of finite groups. Instead of creating a subtype, I decided to make the cardinality function conditional on the class being finite, and then defined it on some representative of the class. Since there is no canonical representative of an isomorphism class, this makes our definition noncomputable. In particular, `quotient.out'` is the noncomputable function that allows us to retrieve an arbitrary element of the isomorphism class.

```
def is_finite_class (C : isomorphism_classes.obj (Cat.of Group)) : Prop :=
quotient.lift_on' C (λ (G : Group), nonempty (fintype G)) ...

noncomputable def class_card (C : isomorphism_classes.obj (Cat.of Group))
  (h : is_finite_class C) : ℕ :=
let G : Group := quotient.out' C, ...

lemma class_card_mk' {G : Group} (hG : fintype G) :
  class_card (quotient.mk' G) (is_finite_class_mk' hG) = @fintype.card G hG :=
-- from src/jordanholder/trivial_class.lean
```

Note that we did not provide a proof that every representative of an isomorphism class has the same cardinality when defining `class_card`. Instead, this happens in `class_card_mk'`, when we show that the cardinality of the class of a group `G` is just the cardinality of `G`.

Equipped with these definitions, we can begin our proof of the fundamental theorem of arithmetic from the Jordan–Hölder theorem. The first step is to build a normal series from any factorisation of a number (not necessarily a prime factorisation). We first construct a general normal embedding between cyclic groups, which we then use at each step of our inductive normal series definition. Note how we have now switched to additive notation.

```
def normal_embedding_to_mul (l : list ℕ) (m : ℕ) (h : 0 < m * l.prod) :
  add_normal_embedding (zmod l.prod) (zmod (m :: l).prod) := ...

def normal_series_of_factors : Π {l : list ℕ} (hl : 0 < l.prod),
  add_normal_series (AddGroup.of (zmod l.prod))
| [] _ := add_normal_series.trivial (by { simp, apply_instance })
| (m::l) hl := add_normal_series.cons
  (AddGroup.of (zmod l.prod)) (AddGroup.of (zmod (m::l).prod))
  (normal_embedding_to_mul l m (by simpa using hl))
  (normal_series_of_factors (pos_right_of_mul_pos (by simpa using hl)))
-- from src/jordanholder/arithmetic.lean
```

Next, we show that each of the factors of this normal series is finite, and their cardinalities are the numbers in the list `l`. In order to access the cardinalities of the factors of the normal series, we use a partial map (pmap), since `add_class_card` is a partial function.

```
lemma finite_factors {l : list ℕ} (hl : l.prod > 0) :
```

```
2    ∀ G ∈ (normal_series_of_factors hl).factors, is_finite_add_class G := ...
3
4  lemma card_factors_of_normal_series_of_factors {l : list ℕ} (hl : 0 < l.prod) :
5    (normal_series_of_factors hl).factors.pmap add_class_card (finite_factors hl)
6    = quotient.mk' l := ...
7  -- from src/jordanholder/arithmetic.lean
```

The last step is to show that if all the numbers in `l` are prime, then the resulting normal series is in fact a composition series. Once we have that, we can prove our corollary.

```
1  def composition_series_of_prime_factors {l : list ℕ} (h : ∀ p ∈ l, nat.prime p) :
2    add_composition_series (AddGroup.of (zmod l.prod)) := ...
3
4  theorem eq_prime_factors {s t : list ℕ} (hs : ∀ p ∈ s, nat.prime p)
5    (ht : ∀ p ∈ t, nat.prime p) : s.prod = t.prod → s ~ t :=
6  have hs' : 0 < s.prod := ...,
7  have ht' : 0 < t.prod := ...,
8  λ h, show setoid.r s t, begin
9    rw [←quotient.eq', ←card_factors_of_normal_series_of_factors hs',
10     ←card_factors_of_normal_series_of_factors ht'],
11   let σ := composition_series_of_prime_factors hs,
12   let τ := composition_series_of_prime_factors ht, ...,
13   have iso : (AddGroup.of (zmod t.prod)) ≃+ (AddGroup.of (zmod s.prod)) := ...,
14   congr' 1, rw [←factors_of_add_equiv iso τ, eq_factors],
15 end
16 -- from src/jordanholder/arithmetic.lean
```

Our statement of the fundamental theorem of arithmetic uses the equivalence relation on lists (~) that establishes that two lists are equivalent if they are permutations of each other. This is in fact the defining relation of the multiset quotient type. Because of this, and because our version of Jordan–Hölder is a statement about multisets, we transform this goal into a statement about the multisets induced by `s` and `t` (lines 8 and 9). We then use the fact that these multisets are equal to the factors of the normal series defined earlier (lines 9 and 10). Since all numbers in the lists are prime, these are in fact the composition series σ and τ (lines 11 and 12). Since these are series of isomorphic groups (line 13), they must have the same factors (line 15).

This concludes the demonstration of a use of our formalisation of the Jordan–Hölder theorem. Of course, the fundamental theorem of arithmetic is already part of the library, and it can be proved without the heavy machinery provided by group theory. It is interesting to see how a corollary that can usually be proved in a few lines in a textbook took nearly two hundred lines of Lean code. If anything, this exposes that our design choices in the formalisation of composition series may not have been optimal. For instance, using the `isomorphism_classes` type completely hides the group that was used to build the isomorphism class, to the point that we cannot retrieve its cardinality in a computable way. This problem is avoided by [14], [21] and [20], since they store the original quotient group in their factors, instead of the isomorphism class. On the other hand, it is sensible to speak of the cardinality of an isomorphism class of groups, and perhaps the problem of defining it so that it can be computed is worth exploring in its own right.

Chapter 5

Conclusion

We are now nearing the end of this project report. Throughout it, we have seen how Lean can be used to formalise mathematics of varying levels of complexity, and the role that each of its features plays in doing so. The result is a suite of formalised solutions to four IMO problems and a development of the theory of composition series of groups.

We have encountered examples of proofs and definitions that virtually mirrored their informal counterparts, such as in question 2 from the 2017 IMO script, or the proof of the second isomorphism theorem. On the other hand, others needed careful consideration of the system, as with the diamond issue in Section 3.4, that greatly detracts from the mathematical content. This leaves Lean in the middle ground in terms of expressiveness and usability. Overall, the formalisation of algebraic problems tends to be the closest to a textbook treatment. Coercions between types are a great tool, because they mostly hide the differences between types that one rarely thinks about when producing informal mathematics. Type classes greatly simplify our job too, in particular when dealing with algebraic structures. The fact that they are mostly hidden can cause some unexpected difficulties, however, such as having to manually modify the instance cache during a proof or dealing with heterogeneous equality.

On the bright side, we have seen that Lean has powerful automation features that can sometimes reduce the work needed, compared to a paper proof. Tactics such as `simp`, `ring` and `dec_trivial` often made bookkeeping activities like algebraic cancellation or checking finitely many individual cases unnecessary. In addition to this, we have witnessed Lean's status as a programming language, by defining computable functions at the same time that we proved their properties. This, which is crucial functionality when formalising software systems, has uses in formalising mathematics too. It allows the user to experiment with constructs that she has defined, backed by the machine's computational power. As such, one could have conjectured that the starting points that gave periodic sequences in question 1 of the 2017 IMO script were the multiples of 3 by repeatedly evaluating the function.

All in all, Lean has proven to be a powerful tool for the formalisation of mathematics. Considering that I first heard about it at the time of starting this project, the fact that I could write down contents from an advanced mathematics course in Lean is evidence of its accessibility. I have found that abstract mathematical constructs can often be expressed

more transparently in Lean than concrete applications. It was harder to work through some of the IMO problems because they involved the application of many different general results and combined the use of several complex types. On the other hand, our formalisation of the Jordan–Hölder theorem was, for the most part, faithful to the informal presentation. Even when new objects needed to be introduced, such as normal embeddings, they were mathematically meaningful and useful on their own. This shows how Lean is capable of expressing a wide range of mathematical concepts and proof strategies. Because of this, one could imagine undergraduates learning material (and even writing assignments) through Lean, especially when dealing with abstract mathematics. Similarly, researchers in mathematics could make a habit of formalising their results in Lean, which would simplify the review process and provide a natural way for some results to depend on others.

In the process of acquiring these insights about proving theorems in Lean, we have not only used *mathlib* but also extended it. Five pull requests have already been merged into their GitHub repository, and still there are lemmas proved along the way that I believe deserve a place in the library. With some refactoring, our formalisation of composition series and the Jordan–Hölder theorem could also be included. This is testimony of how *mathlib* is continuously growing, with new results being added practically every day.

5.1 Further work

There are several ways in which the work presented here can be extended. First and foremost, we have only proved the Jordan–Hölder theorem for finite groups. An obvious extension would be to generalise this to any group. There are also generalisations of the second isomorphism theorem that use results from lattice theory. Combining these would be a vast endeavour and would constitute a substantial addition to *mathlib*. On a similar note, nobody has ever formalised the six problems of an IMO script in a given year. In particular, very little or no work has been done to formalise geometry proofs in Lean.

In addition to further formalisation, it would be interesting to study whether our development of composition series could be used as material for teaching the subject. The advantages would be significant, since a fully formal proof cannot miss out any detail, and thus would arguably be a more exact representation of the mathematical content. Similar work has been done by Kevin Buzzard at Imperial College, where he teaches undergraduates using Lean[1].

There is also the option to extend Lean's automation capabilities. There were numerous tasks during my formalisation efforts that either were repetitive or felt like they should be left to the system. Examples of this are the constant interchange between modular equality and the type of integers modulo n, or having to show explicitly that the quotient by two equal subgroups is the same group. Since Lean is its own metaprogramming language, one could devise new tactics that would help reduce or hide these problems.

Lastly, Lean 4 is now the most recent version. Unfortunately, it is not backwards compatible with Lean 3, and efforts of porting *mathlib* to Lean 4 have only recently began. Once the library is ported, it would be natural to port our developments with it.

[1] He keeps a blog about this on `xenaproject.wordpress.com`.

References

[1] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: 10.1145/321033.321034. URL: https://doi.org/10.1145/321033.321034.

[2] P. C. Gilmore. "A Proof Method for Quantification Theory: Its Justification and Realization". In: *IBM Journal of Research and Development* 4.1 (1960), pp. 28–35.

[3] H. Wang. "Toward Mechanical Mathematics". In: *IBM Journal of Research and Development* 4.1 (1960), pp. 2–22.

[4] James R Guard et al. "Semi-automated mathematics". In: *Automation of Reasoning*. Springer, 1969, pp. 203–216.

[5] William A Howard. "The formulae-as-types notion of construction". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* 44 (1980), pp. 479–490.

[6] Nicolaas Govert de Bruijn. "Automath, a language for mathematics". In: *Automation of Reasoning*. Springer, 1983, pp. 159–200.

[7] Thierry Coquand and Gérard Huet. "The calculus of constructions". PhD thesis. INRIA, 1986.

[8] Deepak Kapur. "Geometry theorem proving using Hilbert's Nullstellensatz". In: *Proceedings of the fifth ACM symposium on Symbolic and algebraic computation*. 1986, pp. 202–208.

[9] Thierry Coquand and Christine Paulin. "Inductively defined types". In: *International Conference on Computer Logic*. Springer. 1988, pp. 50–66.

[10] Bertrand Russell and Alfred North Whitehead. *Principia mathematica*. Vol. 2. Cambridge University Press Cambridge, UK, 1997.

[11] Per Martin-Löf. "An intuitionistic theory of types". In: *Twenty-five years of constructive type theory* 36 (1998), pp. 127–172.

[12] Benjamin Grégoire and Assia Mahboubi. "Proving equalities in a commutative ring done right in Coq". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2005, pp. 98–113.

[13] Sean Wilson and Jacques D Fleuriot. "Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs". In: *Workshop on User Interfaces for Theorem Provers (UITP)*. 2005.

[14] Marco Riccardi. "The Jordan-Hölder Theorem". In: *Formalized Mathematics* 15.2 (2007), pp. 35–51.

[15] Georges Gonthier. "Formal proof – the four-color theorem". In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.

[16] Stephanie Dick. “AfterMath: The work of proof in the age of human-machine collaboration”. In: *Isis* 102.3 (2011), pp. 494–505.

[17] Predrag Janicic, Julien Narboux, and Pedro Quaresma. “The area method: a recapitulation”. In: *Journal of Automated Reasoning* 48.4 (2012), pp. 489–532.

[18] Joseph J Rotman. *An introduction to the theory of groups*. Vol. 148. Springer Science & Business Media, 2012.

[19] Georges Gonthier et al. “A machine-checked proof of the odd order theorem”. In: *International Conference on Interactive Theorem Proving*. Springer. 2013, pp. 163–179.

[20] Assia Mahboubi. “The rooster and the butterflies”. In: *International Conference on Intelligent Computer Mathematics*. Springer. 2013, pp. 1–18.

[21] Jakob von Raumer. “The Jordan-Hölder Theorem”. In: *Archive of Formal Proofs* (Sept. 2014). `https://isa-afp.org/entries/Jordan_Hoelder.html`, Formal proof development. ISSN: 2150-914x.

[22] Leonardo de Moura et al. “The Lean theorem prover (system description)”. In: *International Conference on Automated Deduction*. Springer. 2015, pp. 378–388.

[23] Sander R. Dahmen, Johannes Hölzl, and Robert Y. Lewis. “Formalizing the Solution to the Cap Set Problem”. In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by John Harrison, John O’Leary, and Andrew Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 15:1–15:19. ISBN: 978-3-95977-122-1. DOI: `10.4230/LIPIcs.ITP.2019.15`. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/11070`.

[24] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem proving in Lean*. `https://leanprover.github.io/theorem_proving_in_lean/index.html`. Accessed: 09/04/2021. 2020.

[25] Jeremy Avigad et al. *Mathematics in Lean*. `https://leanprover-community.github.io/mathematics_in_lean/index.html`. Accessed: 09/04/2021. 2020.

[26] Kevin Buzzard, Johan Commelin, and Patrick Massot. “Formalising perfectoid spaces”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Jan. 2020). DOI: `10.1145/3372885.3373830`. URL: `http://dx.doi.org/10.1145/3372885.3373830`.

[27] The mathlib Community. “The lean mathematical library”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Jan. 2020). DOI: `10.1145/3372885.3373824`. URL: `http://dx.doi.org/10.1145/3372885.3373824`.

[28] Jesse Michael Han and Floris van Doorn. “A Formal Proof of the Independence of the Continuum Hypothesis”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 353–366. ISBN: 9781450370974. DOI: `10.1145/3372885.3373826`. URL: `https://doi.org/10.1145/3372885.3373826`.