Proved Algorithms in Geometric Group Theory

A Thesis

submitted to
Indian Institute of Science Education and Research Pune
in partial fulfillment of the requirements for the
BS-MS Dual Degree Programme

by

Anand Rao Tadipatri

Indian Institute of Science Education and Research Pune
Dr. Homi Bhabha Road,
Pashan, Pune 411008, INDIA.

April, 2023

Supervisor: Dr. Siddhartha Gadgil

Certificate

This is to certify that this dissertation entitled Proved Algorithms in Geometric Group Theorytowards the partial fulfilment of the BS-MS dual degree programme at the Indian Institute of Science Education and Research, Pune represents study/work carried out by Anand Rao Tadipatriat Indian Institute of Science Education and Research under the supervision of Dr. Siddhartha Gadgil, Professor, Indian Institute of Science (IISc), Department of Mathematics , during the academic year 2022-2023.

Dr. Siddhartha Gadgil

Committee:

Dr. Siddhartha Gadgil

Dr. Tejas Kalelkar

This thesis is dedicated to the Lean community

Declaration

I hereby declare that the matter embodied in the report entitled Proved Algorithms in Geometric Group Theory are the results of the work carried out by me at the Department of Mathematics, Indian Institute of Science Education and Research, Pune, under the supervision of Dr. Siddhartha Gadgil and the same has not been submitted elsewhere for any other degree.

Anand

Anand Rao Tadipatri

Acknowledgments

Above all, I would like to sincerely thank my advisor Dr. Siddhartha Gadgil not only for introducing me to fascinating cluster of topics emerging around interactive theorem provers and automated theorem proving, but also for providing me with great opportunities to explore and contribute to these areas. I have immensely benefitted from interacting and working with him over the duration of this project and am indebted to him for his advice, guidance and support. I am also grateful for the opportunities this project has provided me with to interact with several computer scientists and mathematicians working on interactive and automated theorem proving.

I am grateful to Dr. Tejas Kalelkar for his valuable guidance as the project expert.

I would also like to thank my parents and family for nurturing my interest in science and supporting me throughout.

I am grateful to Rémi Bottinelli of the Lean community, from whom I learnt a lot about formalising mathematics in Lean's library of formal mathematics `mathlib`. I would also like to express my gratitude to the Lean developers for their remarkably-designed software and the vibrant Lean community for building an impressive set of projects around it.

Abstract

In the recent decades, computers have opened up novel ways of doing mathematics. The speed and raw power of computers has been used to perform searches over previously untractable spaces of objects and proofs, and complementarily, the ability of computers be systematic and methodical has been utilised to formalise and automatically verify complicated proofs.

Although writing programs and constructing proofs seem like different activities, in type-theoretic foundations of mathematics these can both be seen as instances of constructing a term of a type. Proof assistants based on dependent type theory leverage this correspondence to serve both as theorem provers as well as functional programming languages. As programming and proving can be done within the same framework of dependent type theory, computers searches can in principle be augmented with proofs to give "proved algorithms" that have been proved to be correct and returns results together with formal proofs when run.

Geometric group theory is a relatively new area that is at the intersection of algebra and topology, and connected to several others. Over the last few years, there have been some significant results in the area achieved with the help of computer searches.

This thesis is centered on the theme of "Proved Algorithms in Geometric Group Theory", using the *Lean4 theorem prover and programming language* to formally prove results in geometric group theory involving a mix of proofs and computations.

Contents

0.1 Introduction

This thesis is spread across five chapters, each covering different aspects of the broad themes of geometric group theory, formalisation of mathematics and proved computations. The common thread connecting these chapters is the *Lean4 theorem prover and programming language*, which we use for formalisation, programming and meta-programming. The chapters are all independent of each other and can be read in any order.

The first chapter 1 is an introduction to interactive theorem provers and the formalisation of mathematics, with the exposition focused on possible future uses of this technology.

The second chapter 2 describes an original formalisation in Lean4 of Giles Gardam's disproof of Kaplansky's unit conjecture [Gar21], a recent result settling a long-standing conjecture about group rings. The chapter covers the relevant mathematical background, the details of our formalisation, and some of the measures we take to test that the definitions and theorems in our formalisation capture their intended meaning.

The third chapter 3 describes a formalisation of the notion of the *ends* of a graph done jointly with Rémi Bottinelli. The first part of the chapter covers the relevant mathematics, and the second part covers aspects of our formalisation.

The fourth chapter 4 describes Stallings' topological proof of Grushko's theorem – an elegant topological proof of an algebraic result – together with an outline of a possible approach to formalise this result in Lean.

The fifth chapter 5 describes an approach to automatically solve equations in Abelian groups along with an implementation of the idea in Lean4.

0.1.1 Original contribution

The first chapter 1 is primarily expository; however, the section 1.5.5 briefly describes some original work (done jointly with several others) on the automatic formalisation of natural language statements to Lean code using large language models.

The formalisation work described in the second chapter 2 is an original contribution (done

jointly with Dr. Siddhartha Gadgil), describing a real-time formalisation of an important result using a combination of formal proofs and proved algorithms. This work has been submitted to the ITP-2023 conference.

The formalisation of the notion of the ends of a graph and related properties described in the third chapter 3 is original work (done jointly with Rémi Bottinelli). Parts of this code have been merged with the Lean mathematics library `mathlib`.

The idea and the implementation in the fifth chapter 5 are original work.

Chapter 1

Interactive theorem provers and the formalisation of mathematics

Computers have found use in mathematical research for analysing numerical data, performing combinatorial and symbolic manipulation, simulations and interactive visualisation. Programs known as *interactive theorem provers* now make it possible to perform and verify *mathematical reasoning* involving definitions and theorems encoded in a mathematical foundation. This chapter gives an overview of interactive theorem provers 1.1, the benefits they offer1.2 and some of the challenges of formalising mathematics 1.3. The last few sections of this chapter 1.5 2.5 focus on how interactive theorem provers may be integrated with other mathematical software, such as computer algebra systems, visualisation tools, to create unified platforms for experimenting, visualising, creating and verifying mathematics. The examples in this chapter primarily involve the *Lean theorem prover and programming language* [MU21] 1.4, which is also the software used for the work described in the subsequent chapters of this thesis.

1.1 Interactive theorem provers

The notion of a formal logical system – a language and a set of rules for mathematical deduction – can be traced back to Hilbert's Program in the early twentieth century [Zac03], and even earlier to ideas of Leibniz. Various such foundational systems have been pro-

posed since then, most notably Zermelo–Fraenkel set theory with the axiom of choice (ZFC) [Jec03]. Though all known mathematics can be encoded in such a system *in principle*[1], it is exceedingly tedious to do so in practice and is seldom done directly.

A *proof assistant* or *interactive theorem prover* is a computer implementation of a foundational system along with additional layers for translating a high-level description of a mathematical statement or proof given by the user down to the formal details [Hal14]. The resulting formal proof can be mechanically checked for correctness at the axiomatic level, giving a high degree of confidence in its correctness (there is also the question of the trust-worthiness of the program for checking the proof, which is addressed in 1.1.1). Several proof assistants have emerged over the past few decades, based on various different foundations and design principles. Some of the major proof assistants in current use are Isabelle [NWP02], PVS [ORS92], Lean [Mou+15], Coq [Ber08], Agda [Nor09], Mizar [NK09a], Metamath [MW19] and HOL Light [Har09b].

```
/-
# International Mathematical Olympiad 1964, Problem 1b
Prove that there is no positive integer n for which 2^n + 1 is divisible by 7.
-/
theorem imo_1964_q1b : ∀ (n : ℕ), (2 ^ n + 1) % 7 ≠ 0
  | 0 | 1 | 2 => by decide
  | n + 3 => by
    rw [pow_add, Nat.add_mod, Nat.mul_mod, show 2 ^ 3 % 7 = 1 from by rfl]
    simp [imo_1964_q1b n]
```

Listing 1.1.1: An example of a formal proof in Lean

```
variable {R : Type _} [CommSemiring R] {ι : Type _} (s : Finset ι)

theorem degree_prod_le (f : ι → R[X]) :
  (Π i in s, f i).degree ≤ Σ i in s, (f i).degree := by admit
```

Listing 1.1.2: An illustration of the rich syntactic support in modern proof assistants

```
Polynomial.degree_prod_le.{u, w} {R : Type u} {ι : Type w} (s : Finset.{w} ι)
    [inst : CommSemiring.{u} R]
```

[1]Gödel's first incompleteness theorem implies the existence of statements such as the Continuum hypothesis that are independent of the axioms of ZFC [NN88]. However, this does not prevent known mathematical results with informal proofs from being formalised.

```
(f : ι → @Polynomial.{u} R (@CommSemiring.toSemiring.{u} R inst)) :
@LE.le.{0} (WithBot.{0} Nat)
  (@Preorder.toLE.{0} (WithBot.{0} Nat)
    (@WithBot.preorder.{0} Nat
      (@PartialOrder.toPreorder.{0} Nat
  (@StrictOrderedSemiring.toPartialOrder.{0} Nat Nat.strictOrderedSemiring))))
  (@Polynomial.degree.{u} R (@CommSemiring.toSemiring.{u} R inst)
    (@Finset.prod.{u, w} (@Polynomial.{u} R (@CommSemiring.toSemiring.{u} R
  inst)) ι
      (@CommSemiring.toCommMonoid.{u} (@Polynomial.{u} R
  (@CommSemiring.toSemiring.{u} R inst))
        (@Polynomial.commSemiring.{u} R inst))
      s fun (i : ι) => f i))
  (@Finset.sum.{0, w} (WithBot.{0} Nat) ι (@WithBot.addCommMonoid.{0} Nat
  Nat.addCommMonoid) s fun (i : ι) =>
    @Polynomial.degree.{u} R (@CommSemiring.toSemiring.{u} R inst) (f i))
```

Listing 1.1.3: The theorem statement in 1.1.2 translated down to the foundations

With expressive syntax and the facility to build up proofs interactively at a high level, modern proof assistants are beginning to make the large-scale formalisation of mathematics practical [Geu09]. There has been a tremendous amount of recent progress on formalising important mathematical results, leading to the development of large libraries of formal mathematics such Lean's `mathlib` [Com20a], Isabelle's `Archive of Formal Proofs` [MFV21], Coq's `Mathematical Components` libraries [MT17], the `Mizar Mathematical Library` [NK09b] and the `Agda-UniMath` library [RBP+].

Notable formalisations include those of the four-colour problem [Gon+08], the Jordan curve theorem [Hal07], the prime number theorem [Har09a], the Kepler conjecture [Hal+17], the odd order theorem [Gon+13], the central theorem of condensed mathematics [Sch22] and sphere eversion [MDN22]. Large-scale formalisation efforts of this kind can have numerous long-term benefits, some of which are discussed in 1.2. Despite several successes in formalisation and improvements in proof assistants, the endevour remains challenging and tedious; some of these challenges are described in 1.3.

1.1.1 Trusting proof assistants

The most direct benefit of formalising mathematics on a computer is the strong guarantee of correctness that it offers. However, like any large piece of software, a proof assistant is likely to contain bugs in its code. How can one then trust a proof assistant to verify mathematics correctly?

The small trusted kernel

The solution employed in N.G. de Bruijn's `AUTOMATH` [De 94], which has also been incorporated into the design of several modern theorem provers, is to confine proof checking to a small part of the code known as the *kernel*. With this design, a formal proof is guaranteed to be correct if the kernel certifying it is error-free. The kernel is typically a few hundred lines of carefully-written and documented code, open-sourced and reviewed independently by several humans and machines to minimise the chances of errors. The rest of the proof assistant built around the kernel may contain errors that prevent users from proving true statements, but since the proof checking is performed solely by the kernel, these bugs cannot result in an incorrect proof of a statement being accepted. With all the trust concentrated into the small kernel, several layers of untrusted programs can be used in constructing formal proofs without compromising on correctness.

Some proof assistants also support the ability to export proofs in a format that can be independently processed and verified by external tools.

Verifying formalised mathematics

Once a mathematical result has been formalised in a proof assistant, how does one convince a sceptic of its truth? As explained in 1.1.1, it suffices for the sceptic to check the kernel of the proof assistant to be assured of the correctness of proofs formalised in the system. However, to be fully convinced that the result is true, the sceptic must also judge whether the *formal statement* of the theorem, as encoded in the proof assistant, corresponds to the mathematical result being proved. This involves checking the formal statements of the key theorems and their associated definitions, which is significantly easier than verifying the full

proofs themselves. It is a common practice to build library of theorems around each definition – consisting of expected facts and characterising properties – both for future use in proofs involving that definition and for further assurance that the definition captures the intended mathematical meaning. The above measures should give a sceptic strong reasons for believing the correctness of a mathematical result, without ever having to go through its proof. Having a proof format that is human-readable is also useful, since the sceptic can read through the proof and understand the high-level ideas, knowing that the intermediate low-level details have been computer-verified. One may take additional measures to insulate against low-level hardware or software issues, but the increase in certainty that these bring may not be necessary; the fact that the result must have been carefully analysed and understood by a human mathematician in the process of formalising it, combined with the small kernel architecture, already provides certainty which is orders of magnitude higher than traditional peer-review.

1.2 Benefits of formalisation

As formal methods in mathematics and computer science are rapidly growing areas, it is perhaps apt to classify the benefits of formalisation into *immediate benefits* that formalisation already has to offer, and *potential future benefits* that could eventually arise from formalisation. This section primarily deals with the former, and the latter is the focus of 1.5.

1.2.1 Verification of mathematics

Though mathematics maintains a high standard of rigour, flaws and inconsistencies in mathematical arguments are known to crop up from time to time. Formal verification of mathematics using proof assistants offers a way of guaranteeing correctness beyond any reasonable doubt. As noted by Massot [Mas21], formalisation also ensures consistency at both small and large scales. At small scales, formalisation ensures that no minor details or corner cases have been overlooked in the proof. At large scales, it ensures the inter-compatibility of the various definitions and theorems in the library, allowing major refactoring of code to be done easily.

Formalisation is particularly useful for checking the correctness of deeply technical results that are used as *blackboxes*. The recent *Liquid Tensor Experiment* [Sch22] – which originated from Peter Scholze's request to the formalisation communities to help verify some of the technical parts of his work in condensed mathematics – is a notable example of such a formalisation. Formalising the result with a proof assistant helped keep track of the details of the argument, allowed for some drastic simplifications in parts of the proof, and helped Scholze to better understand his result.

Formalisation is also useful for verifying proofs involving a large number of computer calculations which are difficult for humans to comprehend directly. Thomas Hales' *Flyspeck project* [Hal+17] for verifying his proof of the Kepler conjecture and Gonthier's formalisation of the Appel-Haken proof of the four-colour theorem [Gon+08] are notable example of this kind.

1.2.2 Verification of software and hardware

Proof assistants are also used to formally verify the correctness of hardware and software. Hardware errors, such as the Pentium FDIV bug in the Intel Pentium processors [CKZ96], can be hard to detect and expensive to fix; formal verification is now routinely done in the semi-conductor industry to ensure the robustness of hardware. Programs responsible for running safety-critical systems, such as medical devices or railway interlocking systems, are also typically verified using interactive theorem provers; a successful example of this kind of formalisation is the formal verification of the French railway interlocking system by Alstom [PFB19].

1.2.3 Organising mathematical knowledge

Mathematical knowledge is currently spread across several books, papers and articles. Moreover, these sources usually have different notation, conventions and assumptions. In building large libraries of formal mathematics, this knowledge is being systematically organised and preserved in a consistent and compatible way that can be processed by a computer. The careful reflection requires for formalising mathematics not only clarifies existing proofs, but can lead to new and more conceptual proofs of existing results.

Large formal mathematics libraries open up possibilities to build new tools for mathematics research, as discussed in 1.5. Creating such libraries requires large-scale collaboration, and modern software engineering tools such as version control and continuous integration make it possible to collaborate on formalisation without having to read and trust proofs written by others as they are verified by the proof assistant.

1.2.4 Learning and creating mathematics

Proof assistants are also useful tools for *learning* new mathematics. Unlike traditional mathematical texts, in which the level of exposition is fixed by the author, a formal proof can be understood at any level of detail – from just the high-level details of the main statements to the full formal proof with all the logical details specified. Reading proofs with the details spelt out in precise terms and all implementation details visible lends great clarity. Modern editors make it extremely convenient to jump to previously defined concepts and to view the documentation of a definition by hovering. Patrick Massot and collaborators are currently working on tools that translate formal proofs to informal natural language and allow users to dynamically expand and collapse sections of the proof.

Formalising a mathematical result is a good way to understand it deeply. It first requires reformulating the proof into a form where all definitions and concepts are consistent with each other and with the library of mathematics one is building on. Further, it requires choosing suitable ways of representing the mathematical objects under consideration, identifying useful lemmas in the proof, and expanding on the details in a proof.

The process of formalisation can also result in new mathematics, either in the form of major simplifications of existing proofs or novel ideas and abstractions for known results. In the *Liquid Tensor Experiment*, one of the main theorems involved a complicated construction known as the *Breene-Deligne resolution*; by carefully isolating its required properties while formalising the theorem, Johan Commelin was able to replace it with a much simpler construction, now known as the *Commelin complex*. Bourbaki's theory of filters is an example of an abstraction that arose as a rigorous framework for unifying the various kinds of limits in analysis (left limits, limits to infinity, limits of functions at a point, limits of sequences, etc.); this theory is now a standard part of most formal mathematics libraries. Mathematics formalised in a proof assistant can also be further streamlined using *linters*,

which are programs configured to detect details like unused assumptions or identify when results can be stated in greater generality.

1.3 Challenges of formalisation

Despite its numerous benefits 1.2, formalisation of mathematics has not yet seen widespread adoption in the mathematics community. This is primarily because explaining mathematics to a computer currently involves significantly more effort than explaining mathematics to another human, and parts of the formalisation process remain extremely tedious. This section investigates some of the main challenges in formalising mathematics, while also briefly touching upon some possible solutions.

1.3.1 The large gap between informal and formal mathematics

Mathematicians are able to communicate ideas efficiently by conveying details in an incomplete and imprecise way that brings out the high-level ideas and intuitions. Any missing details or minor errors can usually be rectified by other mathematicians understanding the work.

One of the challenges in formalising mathematics, and also designing proof assistants, is to bridge the gap between the high-level ideas and the formal details. The following example from Andrej Bauer's IPAM talk *Formalising Invisible Mathematics* wonderfully illustrates this gap.

> If f is linear, then $f(2 \cdot x + y) = 2 \cdot f(x) + f(y)$.

This statement omits a number of details – that f is a map between vector spaces over a given field, that addition happens in the respective vector spaces, and that 2 refers to the field element and not the natural numbers – most of which can be readily inferred by a reader.

With all the details supplied, the statement looks like

If K is a field, U and V are vector spaces over K, and $f : |U| \to |V|$ is a linear transformation then for all $x, y \in |U|$, $f(2_K \cdot x +_U y) = 2_K \cdot f(x) +_V f(y)$.

Fortunately, modern proof assistants are able to automatically fill in most of these details through a combination of mechanisms known as coercion, typeclass inference and notational scope.

However, in other cases, the gap between formal and informal mathematics persists. Consider the standard elementary proof of the irrationality of the square root of 2, which proceeds by assuming that there is a *rational number* whose square is 2, representing it as the ratio of two *integers*, and finally using the well-foundedness of the *natural numbers* to conclude that the square root of 2 is an irrational *real number*. Though this implicit interaction between the four number systems $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$ and $\mathbb{N}$ can be minimised to some extent by coercions, it is difficult to eliminate entirely.

While the above proof can still be formalised with a few slight changes to the argument, proofs that rely heavily on visual, spatial and numerical intuition are extremely difficult to cast into a sequence of logical deductions [AA11]. Consider the mutilated chessboard problem, which asks whether an 8×8 square board with two opposite corners removed can be tiled by 2×1 dominoes. The standard proof is to colour the board with alternating colours like a chessboard and observe that a 2×1 domino must cover tiles of opposite colours, after which it becomes immediate that the mutilated chessboard cannot be covered by 2×1 dominoes as that would require it to have an equal number of tiles of each colour. Similar difficulties arise in calculating approximate, back-of-the-envelope estimates of quantities using strong numerical intuitions about the nature of various functions, like "$x^4 + 7x + 2$ grows faster than $x^2 + 7$", or "$\frac{sin(x)}{x}$ vanishes at infinity". Such difficulties can perhaps be addressed by developing foundations or abstractions that operate closer to our intuition, combined with powerful automation.

Indeed, there are some successful instances of abstractions and automation bringing formal mathematics closer to mathematical practice. The `ring` tactic in Lean automates trivial calculations like $(a + b)^2 - (a - b)^2 = 4ab$, which would otherwise need a series of repeated invocations of properties like the commutativity of addition and the associativity of multiplication to justify from the ring axioms. Bourbarki's theory of filters provides an elegant abstraction that unified the various kinds of limits that arise in analysis, bypassing the need to adapt theorems about limits to each kind of limit.

1.3.2 The choice of description of mathematical objects

Mathematical objects can be studied from several viewpoints, each offering a different perspective. Moreover, each description has its strengths and weaknesses – for example, some may have good structural properties and be well-suited for proofs, and others may have good computational properties and be better suited for algorithms. The Dedekind construction of the reals may be useful in establishing order-theoretic properties, while the construction via Cauchy sequences may be useful for metric properties. Similarly, natural numbers have unary, binary and decimal representations, each convenient for different purposes.

One of the subtle challenges of formalisation is choosing representations of mathematical objects that are well-suited for the proof and interact well with each other and with the definitions in the library. Indeed, one of the most difficult parts of formalisation, which happens before writing any code, involves expanding the proofs, choosing suitable representations for mathematical objects, disambiguating the notation and reformulating the definitions to bring the result in a form that can be readily verified. It can take several iterations of code before the definitions blend together in a suitable way.

Of course, it is always possible to equivalent definitions of the same mathematical object and use them for different purposes, but this comes with the additional cost of formally proving that these structures are isomorphic. There is also the challenge of showing that results proved for one object hold for an object isomorphic to it, but this problem has an elegant solution due to Voevodsky [Uni13].

As noted by Thurston [Thu94], there is a lot to learn from formalising mathematics as the endeavour can help simplify and clarify mathematics.

1.3.3 Tedious proofs for obvious theorems

Facts which appear completely obvious to a mathematician sometimes need long and detailed justifications, which is another major challenge in formalisation. Mathematical texts often contain a number of minor implicit claims and details that are necessary to state and prove to produce a complete proof. For these reasons, formal proofs are substantially longer than their formal counterparts. The *de Bruijn factor* [Wie00] of a theorem is the ratio of the

length of a formal proof to its informal version, and is usually between 4 and 10. Better automation can help bring down this factor and make formalisation more feasible.

1.3.4 Issues at the foundational level

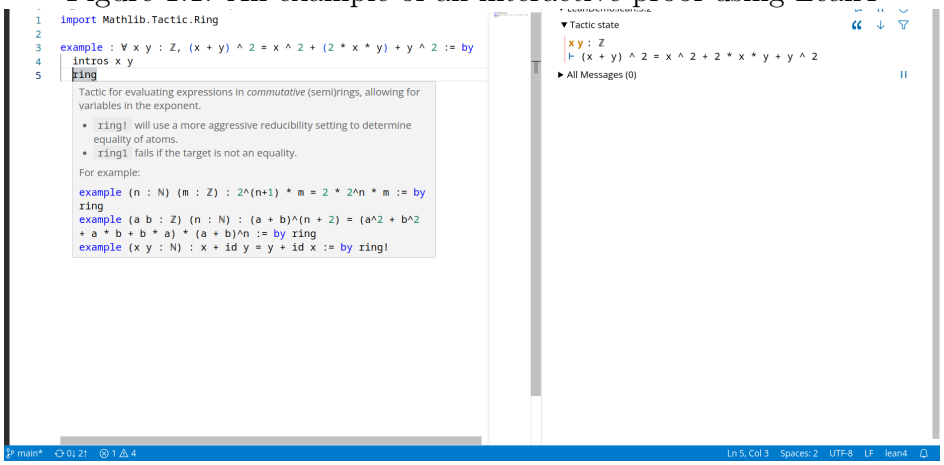
In the process of formalising mathematics, one is encoding definitions and theorems into a logical foundation. Modern foundations are sufficiently expressive that this process usually goes through smoothly, but there are instances where it is substantially more difficult to encode a statement in a theorem prover than it is to write it down on paper. For example, if $v \in \mathbb{R}^{n+m}$ and $w \in \mathbb{R}^{m+n}$, the sum $v + w$ is defined since the vectors belong to the same vector space. However, the vector spaces $\mathbb{R}^{n+m}$ and $\mathbb{R}^{m+n}$ are apriori different to a proof assistant, and additional work is needed to use the fact that $n + m = m + n$ to make the vectors compatible for addition. Preventing such issues may require work at the foundational level, but would have the consequence of making proof assistants easier to use and reduce the learning curve for mathematicians by bringing foundations closer to practice.

1.4 The Lean4 theorem prover and programming language

Lean [MU21] is an interactive theorem prover primarily developed at Microsoft Research by Leonardo de Moura and Sebastian Ullrich. Lean4 is the latest iteration of the Lean theorem prover [Mou+15], re-implemented in Lean itself as an extensible theorem prover and an efficient general-purpose programming language. It features a wide range of programming and meta-programming facilities that can be modified and extended by users, including a parser, elaborator, tactic framework, macro system and code generator. Moreover, one of the largest libraries of formal mathematics – `mathlib` [Com20b] – is written in Lean3 and is steadily being ported to Lean4 (the effort is slightly over half-way through at the time of writing).

Lean4 implements a variant of the mathematical foundations of *dependent type theory* [MS84] known as the *Calculus of Inductive Constructions* [CH86] [Pau15]. Type-theoretic foundations replace the notion of a *set* with the notion of a *type*, which can be regarded either

Figure 1.1: An example of an interactive proof using Lean4



as a collection of related objects (similar to a set) or as a kind of systematic label attached to an object (similar to a data-type in a programming language). By a deep observation known as the *Curry-Howard correspondence* [SU98], an implementation of dependent type theory can be used to encode proofs as well as programs. A particularly illuminating instance of this correspondence is in interpreting the expression `h : P → Q` in type theory, which can be variously understood as a *proof* that P implies Q, a *function* from the type P to the type Q, or a *program* which takes as input a term of type P and returns a term of type Q. The Lean4 theorem prover and programming language takes full advantage of this correspondence by supporting general-purpose programming capabilities such as reading and writing files, executing shell commands and processing strings, in addition to its theorem-proving capabilities. Lean4 can therefore be used to mix proofs and programs in powerful ways, such as writing programs to generate proofs, or writing algorithms that are proved to be correct.

Lean4 is also a highly extensible language, allowing users to define custom notation and implement algorithms for proof automation (as "tactics"). For example 1.4.1 shows an example of Lean's `do` notation, which is a system of macros written in Lean itself to support arbitrary Python-style imperative code. Lean's vast mathematics library `mathlib` also contains several examples of custom syntax, such as in 1.4.2.

Most importantly, Lean4 code is very efficient and performant. As an efficient and extensible general-purpose programming language that is also a theorem prover, Lean4 opens up a wide range of possibilities for mixing programs, meta-programs and proofs within the

same framework.

```
/-- The sum of the squares of the first ‘n‘ natural numbers. -/
def sumOfSquares (n : ℕ) : ℕ := Id.run do
  let mut sum := 0
  for i in [0:n] do
    sum := sum + i^2
  return sum

#eval sumOfSquares 5                -- 30
#eval 0^2 + 1^2 + 2^2 + 3^2 + 4^2   -- 30
```

Listing 1.4.1: An example of `Python`-style imperative code in Lean using the custom `do` notation

```
elab (name := generalizeProofs) "generalize_proofs"
    hs:(ppSpace (colGt binderIdent))* loc:(ppSpace location)? : tactic => do
  let ou := if loc.isSome then
    match expandLocation loc.get with| .wildcard => #[]| .targets t _ => telse
    #[]let fvs ← getFVarIds oulet goal ← getMainGoallet ty ← instantiateMVars (←
    goal.getType)let (_, ⟨_, out⟩) ← GeneralizeProofs.visit ty |>.run.run  nextIdx
    := hs.toList let (_, fvarIds, goal) ← goal.generalizeHyp out fvsfor h in hs,
    fvar in fvarIds dogoal.withContext <| (Expr.fvar
    fvar).addLocalVarInfoForBinderIdent hreplaceMainGoal [goal]
```

Listing 1.4.2: An example of a *tactic* written in Lean

```
@[inherit_doc]postfix:1024 "ˣ" => Units

@[inherit_doc] infix:50 " ∣ " => Dvd.dvd

theorem Monoid.dvd_mul_right [Monoid α] {a b : α} {u : αˣ} : a ∣ b * u ↔ a ∣ b :=
    sorry
```

Listing 1.4.3: An example of Lean's flexible and extensible notation

1.5 Other uses of interactive theorem provers for mathematics

This section discusses some of the uses of interactive theorem provers and related technology for purposes other than the formalisation of mathematics. In particular, the focus is on how interactive theorem provers are situated in the broader context of mathematical software. Most of the projects described in this section are still in their preliminary stages and the content here is largely speculatory.

1.5.1 Exposition

Though formal proofs can be difficult to read directly, informal explanations generated out of formal proofs can offer exciting new ways of communicating mathematics. Tools capable of doing this are under development, and a fascinating demonstration of one such tool can be seen in Patrick Massot's recent IPAM talk or at the following web-page `https://www.imo.universite-paris-saclay.fr/~patrick.massot/Examples/ContinuousFrom.html`. The outputs generated by such a tool include fully hyperlinked proofs that can be expanded and collapsed to any depth, including the foundational details. The notation and reasoning is guaranteed to be unambiguous and correct, as the proof has been verified by the proof assistant. The development of informalisation tools may even influence the way formal proofs are written and give us a better understanding of the relation between formal and informal mathematics. In the future, one can hope for fully-formal mathematics textbooks and research articles that are simultaneously accessible to beginners and experts.

Another direction in using proof assistants for exposition is creating games, such as Kevin Buzzard's Natural Number Game. These are an excellent and engaging way to learn mathematics by proving it on a computer, with the unsolved theorems forming the "levels" of the game.

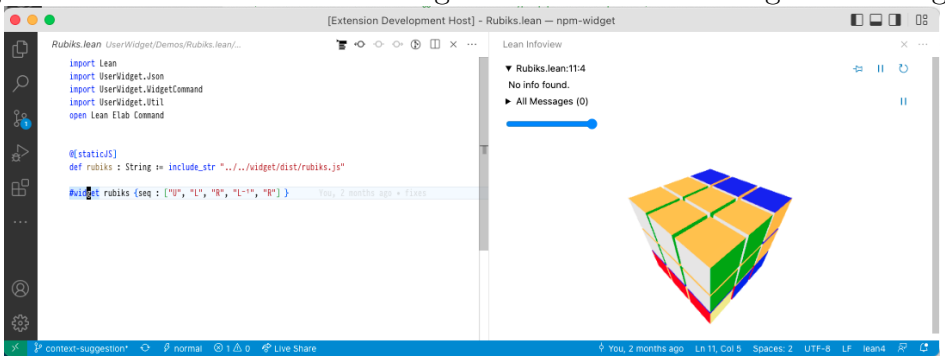
1.5.2 Visualisation

Automatically-generated visualisations accompanying formal proofs are another way in which proof assistants can help in communicating mathematics. There are several freely-available standalone visualisation tools, such as Desmos for plots, Geogebra for 3D visualisation, GraphViz for abstract graphs and networks, and Penrose for generating diagrams from text.

The Lean4 theorem prover features a widget framework that enables the creation of custom user interfaces. As widgets support arbitrary HTML and JavaScript code, it is possible in principle to link with the above-mentioned tools to automatically generate diagrams and visualisations accompanying formal proofs. Visualisation tools in interactive theorem provers may also find use for programming-related tasks such as generating plots, running interactive simulations and visualising the execution of algorithms on specific inputs.

Widgets need not be restricted to only visualising generated images and animations; they may also be used to take as input drawings of commutative diagrams, graphs or knots, and automatically convert them to their formal representation. More generally, widgets could allow users to interact with the mathematics – the SciLean project houses some interesting examples of scientific visualisation with Lean4, some of which allow for user interaction.

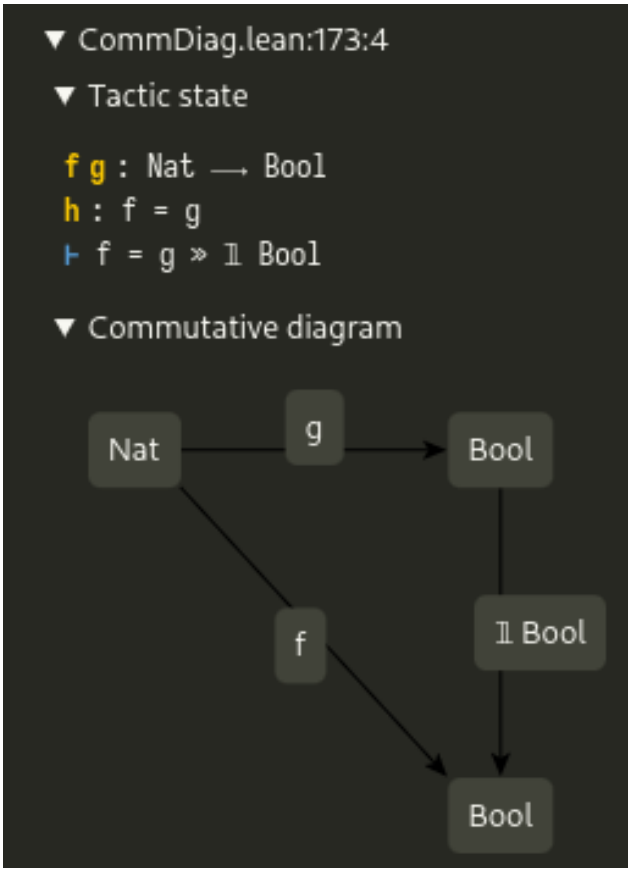
Figure 1.2: An interactive rendering of a Rubik's cube using Lean4 widgets



1.5.3 Verified computation

Computer-algebra systems like SageMath, Maple and Mathematica are valuable tools for mathematical research. However, they have been known to contain bugs and generate incorrect results.

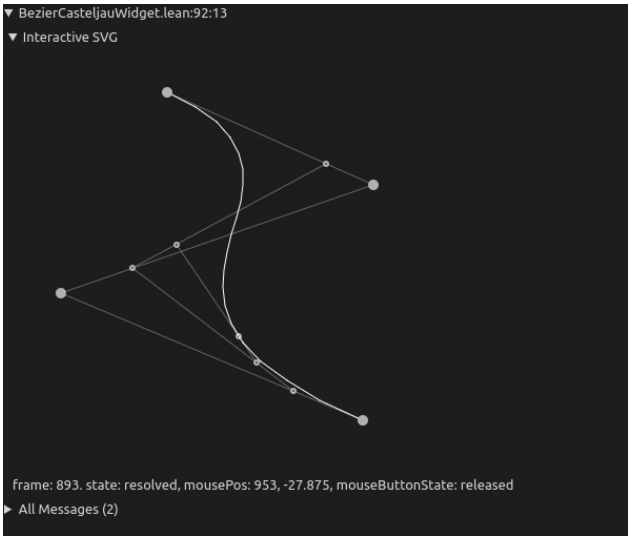
Figure 1.3: Automatic rendering of a commutative diagram from the goal state using Lean4 widgets



Interactive theorem provers equipped with programming capabilities can in principle be used to create *algorithms with proofs* for performing symbolic computations, combining the usefulness of computer-algebra systems with the reliability of proof assistants. A fascinating proof-of-concept is Kaliszyk and Wiedijk's computer algebra system built on top of the HOL Light proof assistant [KW07]. More generally, algorithms can be formalised with proofs of correctness to create libraries of verified computations. An example of such an initiative is [Nip+21].

Formalising symbolic algebra algorithms can have its challenges – the algorithms may involve optimisations that are difficult to verify or make use of complex heuristics. It is also challenging to find representations of mathematical objects that are suitable for both proofs and computations. In such cases, it is still possible to verify the *outputs* of symbolic algebra systems for specific problems after they have been generated – this allows leveraging

Figure 1.4: An interactive visualisation of Bézier curves in `SciLean`



untrusted external tools to perform verified calculations. A similar principle has been used for performing verified reductions for optimisation problems in the Lean4 theorem prover [BMA23].

1.5.4 Automation

Having good proof automation is perhaps the most important requirement for making the formalisation of proofs more practical. There have been a number of experiments in creating programs that link proof assistants with external tools equipped with a large number of heuristics and decision procedures for automating various fragments of mathematics. Some of the most successful examples of automation of this kind are the Sledgehammer tactic of Isabelle [BBN11], and the related Magnushammer tool [Mik+23].

The `aesop` automation tool for Lean4 [LF23] takes a different approach, placing emphasis on being a *whitebox automation tool* which is transparent, predictable, and highly configurable. By choosing suitable theorems from the library and specifying how they are to be applied, Lean4 users can use `aesop` to automate a wide range of obvious proofs.

As automation tools become better, formal proofs may start looking like proof sketches where the high-level details of the proof are supplied by the user and the low-level details are taken care by automation.

1.5.5 Integration with artificial intelligence tools

Recent advances in artificial intelligence, especially the development of large language models [Vas+17], has tremendous potential for formal mathematics. Large language models like ChatGPT are neural networks with several billion parameters trained on a vast amount of text (often significant fractions of the internet) in an unsupervised way. Language models show a number of emergent properties, including the ability to translate text in natural language into computer code. Large language models can be specialised to perform various tasks by *finetuning* on relevant data.

AI chatbots for formal mathematics

A model fine-tuned on data from arXiv, `mathlib` (the Lean mathematics library) and Lean community chat, for example, could function as an AI chatbot specific to formalisation of mathematics in Lean, helping users find relevant concepts in `mathlib`, simplifying proofs and fixing errors.

Autoformalisation of mathematical text

The ability of large language models to automatically translate short mathematical statements into formal code was first observed in the work [Wu+22]. This is done by supplying a few examples of natural language statements with their formal counterparts, followed by just the statement to be translated – this sets the context of the problem, and the model usually continues the text by translating the last statement into formal code. The set of example statements constitutes the *prompt* to the model, and the translation can depend significantly on the prompt used.

In joint work with computer scientists from Microsoft Research, we (the supervisor and the author of this thesis) developed a tool `LeanAIde` to automatically translate short theorem statements to Lean code. It uses a language model internally, but relies on optimisations to the input and output to achieve its success rate. In particular, we customise the input prompt according to the statement to be translated by automatically retrieving pairs of theorems and their docstrings from `mathlib`; once the outputs are generated, they are checked for errors

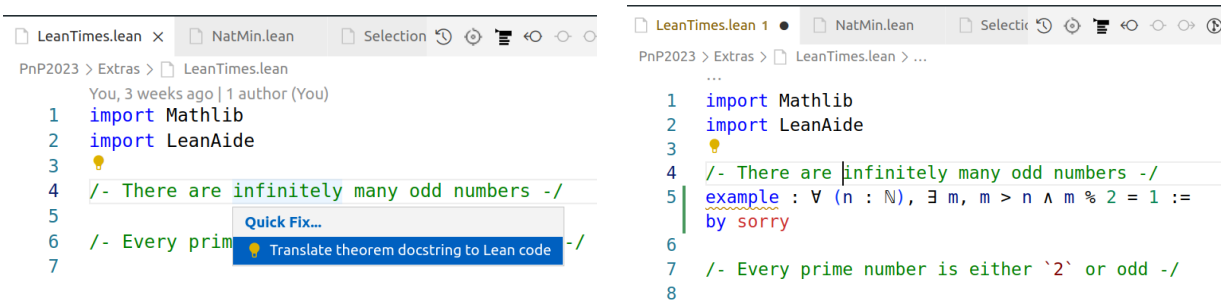


Figure 1.5: A screenshot of `LeanAide` in action, see also `https://youtu.be/IZe3fBZLBR0`

using Lean's efficient internals, automatically processed to rectify minor errors, and only the error-free translations are retained. The tool is available via a convenient interface that is integrated into the user's workflow – on typing the statement of the theorem to be translated as a comment, a lightbulb icon pops up offering to translate the statement to code; once the translation has finished, the translated statement appears below the comment, allowing the user to begin proving the theorem immediately. The work is described further in a note at the 2nd Math-AI Workshop at NeurIPS'22: `https://mathai2022.github.io/papers/17.pdf`.

Another work in a similar direction, but using fixed prompts, is Lean Chat.

Autoformalisation tools capable of formalising larger chunks of text could have a wide impact on the formalisation of mathematics.

Semantic search for mathematics

The mathematics literature is vast, and it would be useful to have tools that are capable of processing this data to help mathematicians and students search for results in the literature. Traditional web-search engines are not well-suited for this task since they are not designed to process mathematical information. Large libraries of formal mathematics are the ideal starting point for such a tool, as they contain mathematical theorems encoded in a way that can be processed and logically manipulated by a computer. Artificial intelligence techniques applied to a corpus of formal mathematics could produce tools capable of finding results that are already in the literature, or even deducing them by combining and specialising known results. Such a tool would be a valuable aid in learning and in mathematical research.

1.6 Conclusion

Interactive theorem provers are powerful tools that hold great promise in changing the way we do mathematics. By integrating with other kinds of mathematical software as described in 1.5, interactive theorem provers may become indispensable tools for mathematicians to experiment, discover, create, automate, verify, communicate and understand mathematics.

The next major steps towards this vision will likely involve a strong mathematical component – advancing foundations to bring formalism closer to mathematical practice, developing strong automation and decision procedures, and reformulating and organising mathematical theories in a way that aligns with both intuition and formalism. Advances in computer science and artificial intelligence will no doubt play a big role too, with collaboration tools, programming language design and automatic program synthesis heavily influencing the design and use of interactive theorem provers. Overall, interactive theorem provers point towards a future of fruitful mathematical collaborations between humans and computers.

Chapter 2

Formalising Giles Gardam's disproof of Kaplansky's unit conjecture

2.1 Overview

The *unit conjecture* is a fundamental and well-known question about group rings (described in 2.3.1) going back to Graham Higman [Hig40] and Irving Kaplansky [Kap70] in the mid-twentieth century. It is a part of a cluster of important conjectures posed by Kaplansky, collectively known as *Kaplansky's conjectures*.

In 2021, Giles Gardam finally settled the unit conjecture in the negative[Gar21]. Gardam's proof was in the form of an explicit counter-example involving a certain well-studied group P (the *Promislow* or *Hantzsche–Wendt* group) and a unit with its inverse in a group ring over P.

This chapter describes our formalisation of Giles Gardam's result in the Lean4 theorem prover[1]. Gardam's result can be viewed as having two components – a *proof* that a certain group P is torsion-free, and a *computation* to verify that the specified elements are units in a group ring over P. Our approach uses a blend of formal proofs and proved computations to verify the result. Our formalisation concluded slightly over a year after Giles Gardam

[1]The documentation for the code can be viewed at `http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture.html`.

announced his result, making it an instance of a *real-time formalisation.*[2]

The relevant mathematical background, including group rings, Kaplansky's conjectures and the structure of the group P, is described in 2.2. The section 2.3 covers the details of our formalisation, with a particular emphasis on our use of Lean4 as both a proof assistant as well as a programming language to integrate the conceptual and computational aspects of the formalisation. We take some extra measures to ensure that our formal definitions satisfy some of their expected mathematical properties (described in 2.4), since incorrect definitions in the statement of the final result are essentially the only source of error in a formalisation. The chapter concludes with some remarks on our formalisation 2.5.

2.2 Mathematical background

2.2.1 Group rings

A *group ring* is an algebraic object associated to a given group and a given ring which occurs naturally in several contexts in mathematics, such as representation theory and algebraic topology. The elements of a group ring can be understood as finite formal linear combinations of elements of the group with coefficients in the ring, similar to polynomials, or better still, Laurent polynomials.

More formally, given a group G and a ring R, the group ring $R[G]$ is the free R-module with basis G, with a multiplication naturally extending that of that of R and G, giving $R[G]$ the structure of a ring and hence an R-algebra. In terms of formal sums, the addition and multiplication can be understood concretely as:

$$\left(\sum_i a_i g_i\right) \cdot \left(\sum_j b_j h_j\right) = \sum_i \sum_j (a_i b_j)(g_i h_j)$$

The group ring $R[G]$ can also be viewed as the set of R-valued functions on G with finite support. Addition of two functions f and g is done point-wise as $x \mapsto f(x) + g(x)$, and

[2]This work has been independently formalised in Lean3: `https://github.com/todbeibrot/counter-example-to-the-unit-conjecture-for-group-rings`

multiplication is given by convolution:

$$x \mapsto \sum_{uv=x} f(u)g(v)$$

which has finite support since f and g have finite support.

The group ring construction $R[-] : \mathbf{Grp} \to R-\mathbf{Alg}$ is left-adjoint to the group of units construction $(-)^\times : R\text{-}\mathbf{Alg} \to \mathbf{Grp}$ which takes an R-algebra to its group of units. The group ring $R[G]$ can also be characterised as the initial object in the category whose objects are R-algebras equipped with a compatible action of G and whose morphisms are G-equivariant R-algebra homomorphisms.

2.2.2 Kaplansky's conjectures

To motivate Kaplansky's conjectures, consider two Laurent polynomials p and q over a field K. The following statements can be easily deduced by considering degrees:

- If $p \cdot q = 0$, then either $p = 0$ or $q = 0$.
- If $p \cdot q = 1$, then both p and q are of the form $c \cdot x^d$, for some $c \in K^\times$ and $d \in \mathbb{Z}$.

Let $K[G]$ be a group ring over a field K. If an element $g \in G$ has *torsion* (i.e., is a non-identity element of G of finite order $n + 1$), then it is possible to find zero divisors in $K[G]$:

$$(1 - g) \cdot (1 + g + g^2 + \ldots + g^n) = 1 - g^{n+1} = 0$$

Moreover, these zero-divisors are *non-trivial*, in the sense that they are not of the form $a \cdot g$ for some $a \in K^\times$ and $g \in G$ (in other words, they have a non-trivial support).

This shows that torsion in group introduces non-trivial zero divisors in the group ring. However, when the group is *torsion-free*, i.e., the only element of finite order is the iden-

tity element, Kaplansky conjectured that the behaviour is identical to the case of Laurent polynomials [Pas76]:

Conjecture 2.2.1 (Kaplansky's zero divisor conjecture)**.** *Let G be a* torsion-free *group and let K be a field. Then the group ring $K[G]$ contains no non-trivial zero divisors.*

Similarly, we know that elements of the form $a \cdot g$, where $a \in K^\times$ and $g \in G$, are units of the group ring – these are the trivial units. It is also known that all but finitely many groups with torsion contain non-trivial units in the group ring over K [Hig40]. Kaplansky's unit conjecture states that when the group G is *torsion-free*, the trivial units are the only units:

Conjecture 2.2.2 (Kaplansky's unit conjecture)**.** *Let G be a* torsion-free *group and let K be a field. Then the group ring $K[G]$ contains no non-trivial units.*

These conjectures, attractive for their simplicity and generality, have remained open for several years. In 2021, Giles Gardam proved the unit conjecture false. The zero divisor conjecture remains open.

2.2.3 Giles Gardam's theorem

Giles Gardam disproved Kaplansky's unit conjecture by explicitly producing a non-trivial unit and its inverse in a certain group ring over the finite field with two elements $\mathbb{F}_2$. The group used in Gardam's counter-example is known in the literature [Pro88][Car14] as the *Promislow group*, *Hantzsche-Wendt group* or the Fibonacci group $F(2,6)$, and is commonly denoted by the letter P. It has the following presentation in terms of generators and relations:

$$\langle a, b | b^{-1}a^2b = a^{-2}, a^{-1}b^2a = b^{-2} \rangle$$

and the elements a^2, b^2 and $(ab)^2$ are commonly denoted by the letters x, y and z respectively. More details about the group P are in the section 2.2.4.

Giles Gardam proved the following theorem [Gar21]:

Theorem 2.2.3 (G. Gardam)**.** *Let P be the group $\langle a, b | b^{-1}a^2b = a^{-2}, a^{-1}b^2a = b^{-2}\rangle$ and set $x = a^2$, $y = b^2$ and $z = (ab)^2$. Set*

$$p = (1 + x)(1 + y)(1 + z^{-1})$$
$$q = x^{-1}y^{-1} + x + y^{-1}z + z$$
$$r = 1 + x + y^{-1}z + xyz$$
$$s = 1 + (x + x^{-1} + y + y^{-1})z^{-1}$$

Then $p + qa + rb + sab$ is a non-trivial unit in the group ring $\mathbb{F}_2[P]$.

Giles Gardam discovered the units with the aid of a computer search using SAT solvers – programs designed to solve instances of the *Boolean satisfiability problem*, which is the problem of determining whether there is an assignment of variables satisfying a Boolean formula built out of Boolean variables and logical operations such as $\vee$, $\wedge$ and $\neg$.

This result was subsequently generalised by Murray to produce non-trivial units in $\mathbb{F}_d[P]$, where d is an arbitrary prime number [Mur21].

2.2.4 The group P

The Promislow group P was first identified by Promislow in [Pro88] as an example of a torsion-free group that does not satisfy a certain group-theoretic property known to imply the unit conjecture for group rings over that group. The group can be described in terms of generators and relations as follows

$$\langle a, b | b^{-1}a^2b = a^{-2}, a^{-1}b^2a = b^{-2}\rangle$$

Following Gardam [Gar21], we introduce two new variables $x = a^2$ and $y = b^2$ into the presentation

$$\langle a, b, x, y | b^{-1}xb = x^{-1}, a^{-1}ya = y^{-1}, x = a^2, y = b^2 \rangle$$

to express the group as the amalgam of two Klein bottle groups $\langle x, b | b^{-1}xb = x^{-1} \rangle$ and $\langle y, a | a^{-1}ya = y^{-1} \rangle$, along the isomorphic subgroups $\langle x, b^2 \rangle \cong \langle a^2, y \rangle$. The subgroups are in fact isomorphic to $\mathbb{Z}^2$, the free Abelian group on two generators. Being an index-2 subgroup of each copy of the Klein bottle group, it is a normal subgroup of both groups and hence a normal subgroup in the amalgam. The corresponding quotient is isomorphic to $\mathbb{Z}/2 * \mathbb{Z}/2$, which is the infinite dihedral group D_∞. The commutator of the infinite dihedral group is an infinite cyclic group $[D_\infty, D_\infty]$ generated by the image of $abab \in P$ in D_∞, making the corresponding quotient isomorphic to the Klein Four group $\mathbb{Z}/2 \times \mathbb{Z}/2$. Labelling the element $abab$ as z, we see that

$$\begin{aligned} z^{-1}xz = (abab)^{-1}a^2(abab) &= (ab)^{-1}(b^{-1}a^2b)(ab) \\ &= (ab)^{-1}(a^{-2})(ab) = x \end{aligned}$$

Similarly,

$$\begin{aligned} z^{-1}yz = (abab)^{-1}(b^2)(abab) &= (ab)^{-1}(b^{-1}(a^{-1}b^2a)b)(ab) \\ &= (ab)^{-1}b^{-2}(ab) = b^{-1}(a^{-1}b^{-2}a)b = y \end{aligned}$$

This shows that z commutes with x and y. The map $\pi : P \to \mathbb{Z}/2 \times \mathbb{Z}/2$ obtained by composing the intermediate maps through D_∞ has as kernel $\langle x, y, z \rangle$ – which is in fact isomorphic to $\mathbb{Z}^3$ – thus making P the centre of a short exact sequence of groups

$$1 \to \mathbb{Z}^3 \to P \to \mathbb{Z}/2 \times \mathbb{Z}/2 \to 1$$

The group P is therefore a *metabelian group* – a group containing an abelian normal subgroup such that the quotient is also abelian. The kernel and quotient of the above short exact sequence are well-understood abelian groups – one is a finitely-generated free abelian group and the other is a finite abelian group. It would be useful to find a description of P

completely in terms of these two well-understood groups and some additional data. More generally, given abelian groups A and B, we would like to describe an arbitary metabelian group corresponding to A and B in terms of some explicit data.

Group extensions and factor systems

A group G is defined to be an *extension* of Q by K if there is a short exact sequence of groups

$$1 \to K \to G \to Q \to 1$$

The *group extension problem* is to classify all extensions G for a given Q and K. We are interested in the special case of the group extension problem when Q and K are both abelian.

Consider an arbitrary extension G of Q by K, where Q and K are abelian. As a set, G can always be identified with $K \times Q$ by choosing a *section*, i.e., a function $\sigma : Q \to G$ such that $\forall q \in Q, \pi(\sigma(q)) = q$, where $\pi : G \to Q$ is the map occurring in the exact sequence. Every element of G can be *uniquely* written in the form $k \cdot \sigma(q)$, where $k \in K$ and $q \in Q$.

The product of two elements $k \cdot \sigma(q)$ and $k' \cdot \sigma(q')$ works out to be

$$\begin{aligned}(k \cdot \sigma(q)) \cdot (k' \cdot \sigma(q')) &= (k \cdot \sigma(q)) \cdot k' \cdot (\sigma(q)^{-1} \cdot \sigma(q) \cdot \sigma(q')) \\ &= (k \cdot (q \bullet k')) \cdot (\sigma(q) \cdot \sigma(q'))\end{aligned}$$

where $q \bullet k' = \sigma(q) \cdot k' \cdot \sigma(q)^{-1}$ denotes the conjugation action of Q on K. The well-definedness of the action uses the fact that K is abelian and hence the action of Q on K does not depend on the section σ chosen.

Since σ is an arbitary map and not necessarily a homomorphism, it is not guaranteed that $\sigma(q) \cdot \sigma(q')$ is equal to $\sigma(q + q')$. The extent of this deviation is captured by the *cocycle* $c : Q \times Q \to G$, defined as $c(q, q') = \sigma(q) \cdot \sigma(q') \cdot \sigma(q \cdot q')^{-1}$. It turns out that the cocycle always takes values in the kernel K, since $\pi(c(q, q')) = \pi(\sigma(q) \cdot \sigma(q') \cdot \sigma(q \cdot q')^{-1}) = q \cdot q' \cdot (q \cdot q')^{-1} = 1$,

using the fact that σ is a section of the homomorphism π. Thus we have

$$(k \cdot \sigma(q)) \cdot (k' \cdot \sigma(q')) = (k \cdot (q \bullet k') \cdot c(q, q')) \cdot \sigma(q + q')$$

On the set $K \times Q$, the above formula translates to

$$(k, q) \cdot (k', q') = (k + q \bullet k' + c(q, q'), q + q')$$

using additive notation for the abelian groups K and Q. This shows that a group extension of Q by K determines an action of Q on K, and a section $\sigma : Q \to G$ determines a cocycle $c : Q \times Q \to K$. These together give a product structure on the set $K \times Q$. When the section is a homomorphism, the cocycle $c : Q \to Q \to K$ is uniformly the identity, reducing this to the familiar semi-direct product. The associativity of the multiplication on G implies the following condition on c, known as the cocycle condition:

$$\forall qq'q'' \in Q, c(q, q') + c(q + q', q'') = q \bullet c(q', q'') + c(q, q' + q'')$$

Functions $c : Q \times Q \to K$ satisfying this condition are known as *cocycles*. In the other direction, given abelian groups Q and K, an action of Q on K by automorphisms and a function $c : Q \times Q \to K$ satisfying the above condition, we can define a group structure on $K \times Q$ with the following multiplication

$$(k, q) \cdot (k', q') = (k + q \bullet k' + c(q, q'), q + q')$$

In fact, all metabelian groups are of this form. Given a section $\sigma : Q \to G$ and a function $\phi : Q \to K$, the function $\sigma' : Q \to G$ defined as $q \mapsto \phi(q) \cdot \sigma(q)$ is also a section, and the cocycle c' computed from σ' differs from the cocycle c computed from σ by a so-called *co-boundary*. Thus any metabelian group can be completely described by abelian groups Q and K, together with an action of Q on K by automorphisms and a cocycle $c : Q \times Q \to K$ up to a coboundary.

Returning to the original group P, we can choose the section $\sigma : \mathbb{Z}/2 \times \mathbb{Z}/2 \to P$

taking the values $\{1, a, b, ab\}$. In the rest of this chapter, Q and K are used to denote the subgroups $\mathbb{Z}/2 \times \mathbb{Z}/2$ and $\mathbb{Z}^3$ of P. We can use the explicit group action of $\mathbb{Z}/2 \times \mathbb{Z}/2$ on $\mathbb{Z}^3$ by automorphisms and the cocycle $c : (\mathbb{Z}/2 \times \mathbb{Z}/2) \times (\mathbb{Z}/2 \times \mathbb{Z}/2) \to \mathbb{Z}^3$ determined by σ to construct P as a group extension with the underlying set $\mathbb{Z}^3 \times (\mathbb{Z}/2 \times \mathbb{Z}/2)$ and multiplication determined by the action and cocycle. This is the approach we take in the formalisation 2.3.4, as this description of P has the desired structural and computational properties.

The torsion-freeness of P

The following is a sketch of the proof that the group P is torsion-free.

Suppose $g \in P$ and $n \in \mathbb{N}$ are such that g is a torsion element with exponent $n+1$, i.e., such that $g^{n+1} = 1$ (we choose $n+1$ rather than n in the exponent since $g^0 = 1$ is true for all group elements). Observe that $(g^2)^{n+1} = g^{2(n+1)} = (g^{n+1})^2 = 1^2 = 1$, i.e., g^2 is also a torsion element with the same exponent $n+1$. Using the explicit description of the group P, we can deduce that $g^2 \in \mathbb{Z}^3$. Since $\mathbb{Z}^3$ is torsion-free, it follows that g^2 must be 1. Using the explicit description of P again, we conclude that $g = 1$, i.e., the group P is torsion-free. ■

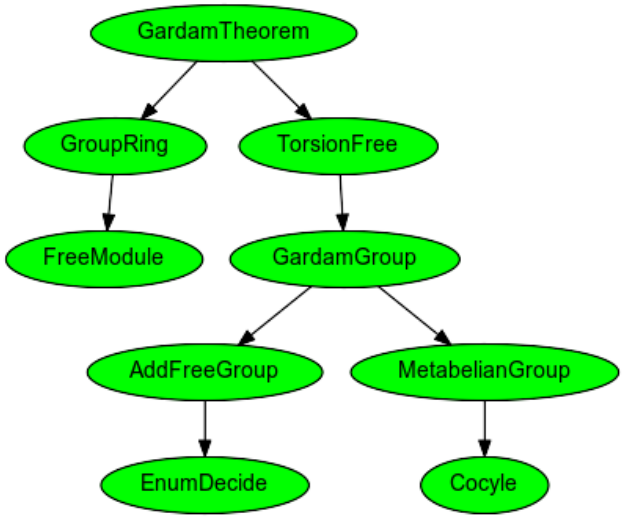
2.3 Formalisation

The full proof of Giles Gardam's result comprises two components – a *proof* that the group P is torsion-free, and a *computation* in the group ring $\mathbb{F}_2[P]$ to confirm that the elements identified by Gardam are (non-trivial) units. We use the nature of Lean4 as both a proof assistant and programming language to seamlessly integrate the conceptual and computational aspects in our formalisation.

We also make use of automation in two distinct forms - *typeclasses* and *tactics*.

- **Typeclasses** Typeclasses in Lean [SUM20] are a framework for recording, specifying, synthesising and composing facts about data automatically through a mechanism known as *typeclass inference*. The `Decidable` typeclass of Lean acts as a bridge between proofs and computations. A proposition `P` is *decidable* if it can be algorithmically settled in the positive or negative. Familiar examples of decidable problems include the

Figure 2.1: A (forward) dependency graph for the formalisation



equality of natural numbers or graph connectivity; equality of functions of the type $\mathbb{N} \to$ `Bool` is not decidable as it involves checking infinitely many values. For a proposition codeP, an instance of *Decidable* P can be specified by providing an algorithm to mechanically settle it in finite time. Given a concrete decidable proposition `P`, typeclass inference attempts to automatically synthesise a decision procedure by specialising and combining existing decision procedures. Typeclasses thus provide a powerful form of automation.

- **Tactics** Lean proofs are generally written interactively by supplying a sequence of *tactics*, which are short snippets of text that modify the goal. Being a customisable language, Lean allows users to modify existing tactics or write their own from scratch. A notable general-purpose automation tactic heavily used in this formalisation is `aesop`, which has been briefly described in 1.5.4.

2.3.1 Group rings

We define group rings in Lean keeping in mind the following goals:

- It should be possible to algorithmically decide when two elements of the group ring $R[G]$ are equal (assuming it is already possible to do so in the group and the ring).

- It should be possible to define R-module homomorphisms from the group ring $R[G]$ by extending functions on G.

The first is a computational goal, and the second is a conceptual one. To achieve both these goals, we first define *free modules*[3] as quotients of formal sums 2.3.1; the equivalence relation on formal sums can be described in two equivalent ways, each suited for a different goal. Group rings are finally defined in terms of free modules.

Formal sums and coordinates

Fix a ring R and a set X. An expression of the form $\sum_{i=0}^{n} a_i x_i$, where $a_i \in R$ and $x_i \in X$, describes an element of the free module $R[X]$. Such an expression can be represented as a list of ordered pairs (a_i, x_i), or equivalently as a term of type `List (R × X)`. An element of the free module $R[X]$ can be described by more than one such expression – for example, if $a, b \in R$ and $x, y \in X$, $ax + by$ and $by + ax$ define the same element of the free module $R[X]$. The free module can be defined as a quotient of the set of formal sums `List (R × X)` by a suitable equivalence relation.

Given a formal sum $s = \sum_{i=0}^{n} a_i x_i$, we can associate to it a *coordinate function* $\chi_s : X \to R$, with the coordinate of an element x of X being the sum of the coefficients a_i corresponding to indices i with $x_i = x$. Observe that only finitely many coordinates are non-zero, and that computing the coordinate of an element requires decidability of equality in X.

We can now define the free module $R[X]$ as the quotient of formal sums by the equivalence relation given by equality of coordinate functions.

Supports and decidable equality

When the basis X is infinite, it is impossible to decide the equality of the coordinate functions χ_{s_1} and χ_{s_2} of two formal sums s_1 and s_2 by enumerating on the basis. However, it is possible to decide equality using the *support* of a formal sum $s = \sum_{i=0}^{n} a_i x_i$, which is defined to be the set of elements x_i occurring in the sum. This is a coarse notion of support, and contains the

[3] `http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture/FreeModule.html`

actual support – the set of elements on which the coordinate function is non-zero (formally stated in Listing 2.3.1).

```
theorem nonzero_coord_in_support
  (s : FormalSum R X) : ∀ x : X, 0 ≠ s.coords x → x ∈ s.support := ...
```

Listing 2.3.1: Non-zero coordinates in support

It follows that the coordinates of a formal sum $s = \sum_{i=0}^{n} a_i x_i$ are zero everywhere outside the support, $\text{supp}(s)$ and thus to compare the coordinates of two formal sums s_1 and s_2, it suffices to compare their values on the union of the two supports $\text{supp}(s_1)$ and $\text{supp}(s_2)$. Since supports of formal sums are always finite, this is a check on a finite set and can be done algorithmically even if the basis X itself is infinite. This shows the decidability of equality in $R[X]$, through the notion of supports of formal sums.

Elementary moves and universal properties

It is useful to be able to define R-module homomorphisms from $R[X]$ by extending functions on X, for example to construct the multiplication function in the group ring. It is straightforward to define an extension on formal sums by specifying $f(\sum_{i=0}^{n} a_i x_i) = \sum_{i=0}^{n} a_i f(x_i)$, but a different description of the equivalence relation is more convenient to show that this is well defined in the quotient $R[X]$. We define a relation on formal sums given by the following *elementary moves*:

- if the first term has zero coefficient, it is deleted.
- if the first two terms are of the form `(a, x)` and `(b, x)` for some `x : X`, they are replaced by a single term `(a + b, x)`.
- the first two terms are exchanged.
- a term is prepended to two formal sums related by an elementary move.

This is not an equivalence relation, but the quotient in Lean is defined for the equivalence relation generated by it.

It is straightforward to show that coordinates are unchanged by elementary moves. The key result, which takes some work, is to show that two formal sums with the same coordinate function are related by a sequence of elementary moves, i.e., have equal images in the quotient by elementary moves.

This in turn depends on the technical result that if a formal sum s has $a_0 := \chi_s(x_0) \neq 0$ for some $x_0 \in X$, then s is related by a sequence of elementary moves to a formal sum of the form $a_0 x_0 + t$, with the number of terms in t less than the number in s (stated in Listing 2.3.2), which is proved by well-founded recursion.

```
theorem nonzero_coeff_has_complement  (x₀ : X)(s : FormalSum R X) :
  0 ≠ s.coords x₀ →
  (∃ ys : FormalSum R X,
    (((s.coords x₀, x₀) :: ys) ≃ s) ∧ (List.length ys < s.length)) := ...
```

Listing 2.3.2: Complements in formal sums

This is used to show that coordinate functions and elementary moves both define the same equivalence relation on formal sums, and the two descriptions of the free module $R[X]$ are used in proving both the computational and conceptual properties needed.

Group rings

Group rings are defined from formal sums [4] by constructing a multiplication function on formal sums, first by right multiplication by a *monomial* $a \cdot g$, and then recursively for arbitrary formal sums. Various properties are proved by (iterated) induction. These allow us to define the product on $R[G]$ by showing invariance under elementary moves.

2.3.2 Enumeration and decision procedures

The construction of P as a metabelian group from an action and a cocycle requires supplying the relevant functions together with proofs that they satisfy the required properties. These verifications are examples of details that are typically omitted in mathematical practice but

[4] http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture/GroupRing.html

are nevertheless required for a fully formal proof. This section describes our approach to automating these verifications using decision procedures and enumeration. [5]

Enumeration

If a property P can be automatically checked (or *decided*) for any given element of a set X, and the set X itself is finite, then the property $\forall x \in X, P(x)$ can also be checked algorithmically by enumeration on X. This idea, when applied iteratively in conjunction with Lean's typeclass inference mechanism, allows us to expand the collection of statements that can be automatically decided.

The property of a type being "exhaustively checkable" in the sense described above is captured by the `DecideForall` typeclass (see listing 2.3.3).

```
/-- A typeclass for "exhaustively verifiable types", i.e.,
  types for which it is possible to decide whether a given (decidable) predicate
   holds for all its elements. -/
class DecideForall (α : Type _) where
  decideForall (p : α → Prop) [DecidablePred p]:
    Decidable (∀ x : α, p x)
```

Listing 2.3.3: The `DecideForall` typeclass

We prove some useful facts about `DecideForall` and exhibit various ways to construct such types (through products, direct sums and functions), which are registered as *typeclass instances*. These can then be used to automate some routine verifications, such as in Listing 2.3.4, which proves associativity of addition in $\mathbb{Z}/3\mathbb{Z}$.

```
theorem Zmod3.assoc :
   ∀ x y z : Fin 3, (x + y) + z = x + (y + z) := by decide
```

Listing 2.3.4: Associativity by enumeration

The same underlying ideas are used to verify the cocycle condition in the construction of P.

[5] `http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture/EnumDecide.html`

Remark 2.3.1. Somewhat paradoxically, there are also examples of infinite sets that can be exhaustively checked. Moreover, these are characterised as those that are topologically compact (under the so-called *intrinsic topology*). These are described further in Martín Escardó's paper [Esc08].

Enumeration of basis of free Abelian groups

Verifying whether a map from a group to the set of endomorphisms of another group is a group action by automorphisms requires checking equality of homomorphisms. When the target group is infinite (which is the case in the construction of P, where the target group is $\mathbb{Z}^3$), an automatic check by enumeration is no longer possible.

However, since $\mathbb{Z}^3$ is a finitely generated free Abelian group, any homomorphism from it is determined entirely by the values at the three basis elements $(1,0,0), (0,1,0), (0,0,1)$. As this is a finite set, it is still possible in this case to check equality of homomorphisms by enumerating the basis.

To this end, we define free Abelian groups equipped with a basis as a typeclass `AddFreeGroup`, and prove basic proprties including that $\mathbb{Z}$ is a free Abelian group on the singleton type and that the product of free groups is free on the direct sum of the bases. We also show that when the basis of the free Abelian group has an instance of `DecideForall`, equality of homomorphisms is determined by equality on the basis. This is the crucial result that allows us to automatically verify that the function `P.action` in the construction of P is in fact an action of Q on K by automorphisms.

2.3.3 Construction of metabelian groups

We perform the general construction of metabelian groups in Lean [6] as group extensions following the description in 2.2.4.

We begin by defining cocycles and group actions by automomorphisms using typeclasses, which is the standard way of encoding algebraic structures in `mathlib`. As a convention, we take cocycles to be *normalised*, i.e., when both inputs of the cocycle are the identity element,

[6] `http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture/MetabelianGroup.html`

the value of the cocycle is the identity element of the target group. For convenience, we also make a few standard deductions from the cocycle condition and annotate these lemmas with the `aesop` attribute.

```
class Cocycle {Q K : Type _} [AddGroup Q] [AddGroup K] (c : Q → Q → K) where
 /-- An action of the quotient on the kernel by automorphisms. -/
 α : Q → (K →+ K)
 /-- A typeclass instance for the action by automorphisms. -/
 autAct : AutAction α
 /-- The value of the cocycle is zero when its inputs are zero, as a convention.
  -/
 cocycle_zero : c 0 0 = (0 : K)
 /-- The *cocycle condition*. -/
 cocycle_condition : ∀ q q' q'' : Q, c q q' + c (q + q') q'' = q +ᵥ c q' q'' + c
  q (q' + q'')
```

Listing 2.3.5: The `Cocycle` typeclass

As outlined in 2.2.4, any metabelian extension of an Abelian group K by another Abelian group Q determined by an action α and cocycle c can be constructed on the underlying set $K \times Q$, with the multiplication given by the formula in Listing2.3.6.

```
def mul : (K × Q) → (K × Q) → (K × Q)
  | (k, q), (k', q') => (k + (q +ᵥ k') + c q q', q + q')
```

Listing 2.3.6: The formula for multiplication in a Metabelian group

The identity and inverse operations of the group can likewise be described by explicit formulae. The various properties required for proving the group structure, such as cancellation of inverses and associativity of multiplication, are proved (and to a large extent automated) in Lean's tactic proof mode using the `aesop` tactic.

As incorrect definitions can be a failure point in formalisation, we also verify that the resulting group lies in an exact sequence flanked by K on the left and Q on the right, though this fact is not used anywhere in the code.

2.3.4 The construction of the group P

The group P can be constructed as a metabelian group with kernel $K := \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and quotient $Q := \mathbb{Z}/2 \times \mathbb{Z}/2$ [7] using the explicit action and cocycle given in Giles Gardam's paper [Gar21]. These are given in Listings 2.3.7 and 2.3.8. The three generators of the free group K are given the labels x, y, z , and the names e, a, b, c refer to the four elements of Q.

```
/-- A temporary notation for easily describing products of additive monoid
    homomorphisms. -/
local infixr:100 " × " => AddMonoidHom.prodMap

/-- The action of ‘Q‘ on ‘K‘ by automorphisms.
The action can be given a component-wise description in terms of ‘id‘ and ‘neg‘,
    the identity and negation homomorphisms. -/
def action : Q → (K →+ K)
  | .e =>  .id ℤ  ×  .id ℤ  ×  .id ℤ
  | .a =>  .id ℤ  ×  neg ℤ  ×  neg ℤ
  | .b =>  neg ℤ  ×  .id ℤ  ×  neg ℤ
  | .c =>  neg ℤ  ×  neg ℤ  ×  .id ℤ
```

Listing 2.3.7: The action of Q on K by automorphisms

```
/-- The cocycle in the construction of ‘P‘. -/
def cocycle : Q → Q → K
  | a , a  => x
  | a , c  => x
  | b , b  => y
  | c , b  => -y
  | c , c  => z
  | b , c  => -x + -z
  | c , a  => -y + z
  | b , a  => -x + y + -z
  | _ , _  => 0
```

Listing 2.3.8: The cocycle

[7] http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture/GardamGroup.html

Performing the construction also requires proofs that these functions indeed determine an action and a cocycle respectively. The two groups involved in the construction have descriptions which are particularly amenable to computation – the group K is the free abelian group on three generators, and the group Q is a finite, non-cyclic group with four elements. Using the decision procedures developed in 2.3.2, these verifications can be automated.

The group P is finally constructed as a metabelian group using the general construction of 2.3.3 by supplying the specific action and cocycle.

```
def P := K × Q

instance (priority := high) PGrp : Group P := MetabelianGroup.metabelianGroup
    cocycle

instance : DecidableEq P := inferInstanceAs <| DecidableEq (K × Q)
```

Listing 2.3.9: The group P

A consequence of having the group structure defined on the set $\mathbb{Z}^3 \times (\mathbb{Z}/2 \times \mathbb{Z}/2)$ is that each group element can be written uniquely as a product and equality of group elements is decidable since equality of each of the components is decidable. This is a computational advantage of the metabelian description of the group P.

Remark 2.3.2. The set $P := K \times Q$ also admits a product group structure, in addition to the specified metabelian group structure. To resolve this ambiguity, we set the `priority` for the metabelian group instance higher. For general code hygiene, we mark instances involving the type $K \times Q$ as `scoped`, so that these are confined to the namespace in which they are defined.

2.3.5 The torsion-freeness of P

The proof of torsion-freeness of the group P follows the mathematical sketch in 2.2.4. The property of torsion-freeness is once again implemented as a typeclass in our code. [8]

As a first step, we need to prove that the square of any element of P lies in the subgroup K. This is done in two steps – by first constructing a map $sq : P \to K$ and then proving

[8] `http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture/TorsionFree.html`

that it (more precisely, its inclusion) agrees with the function taking an element of P to its square. This proof is a direct consequence of unfolding the definitions, and can be done automatically using `aesop`.

The torsion-freeness of the kernel K is deduced by typeclass-inference from general the general facts that the group of integers $\mathbb{Z}$ is torsion-free and that the product of torsion-free groups is torsion-free.

The third step in the proof is to show that the group P does not contain any elements of order 2. The structure of the group P makes it convenient to prove this fact by taking cases on the second component of a given group element and simplifying with `aesop`.

Together with the general fact that if $g^n = 1$ for some $g \in P$ and $n \in \mathbb{N}$ then $(g^2)^n = 1$, the proof of torsion-freeness of P follows: For suppose $g^n = 1$, where $g \in P$ and $n > 1$. Then the identity $(g^2)^n = 1$ can be deduced from general properties of exponentiation. But $g^2 \in K$, which is a torsion-free group. Hence $g^2 = 1$. But the group P has no elements of order 2, which forces $g = 1$. Thus the only element of P with non-trivial torsion is the identity element. ■

2.3.6 Gardam's theorem

We finally come to the formal verification of Giles Gardam's counter-example. [9] The unit conjecture is stated as follows in our code:

```
/-- The statement of Kaplansky's Unit Conjecture:
The only units in a group ring, when the group is torsion-free and the ring is a
    field, are the trivial units. -/
def UnitConjecture : Prop :=
  ∀ {F : Type _} [Field F] [DecidableEq F]
  {G : Type _} [Group G] [DecidableEq G] [TorsionFree G],
  ∀ u : (F[G])ˣ, trivialNonZeroElem (u : F[G])
```

This relies on `trivialNonZeroElem`, which is defined as the property of a free module element having a unique non-zero coordinate.

[9] `http://math.iisc.ac.in/~gadgil/unit_conjecture/UnitConjecture/GardamTheorem.html`

The occurrences of `DecidableEq` in our definition reflect our decision to work constructively. If one were to work classically, where every proposition is decidable, then the hypotheses of decidable equality could be omitted.

Giles Gardam's unit α is assembled from four pieces p, q, r and s. Likewise, its inverse α' is assembled from four pieces p', q', r' and s'.

To prove that α is non-trivial, we exhibit two distinct coordinates on which it takes non-zero values.

To verify that α is a unit, we prove the proposition $\alpha \cdot \alpha' = 1$ using the `native_decide` tactic. The way in which the group ring machinery was set up in our code makes this proposition decidable.

The final proof of Giles Gardam's theorem, that the Unit conjecture is false, follows as a simple consequence.

```
/-- A proof of the existence of a non-trivial unit in `𝔽₂[P]`. -/
def Counterexample : {u : (𝔽₂[P])ˣ // ¬(trivialNonZeroElem u.val)} :=
⟨⟨α, α', by native_decide, by native_decide⟩, α_nonTrivial⟩

/-- Giles Gardam's result - Kaplansky's Unit Conjecture is false. -/
theorem GardamTheorem : ¬ UnitConjecture :=
  fun conjecture => Counterexample.prop <|
  conjecture (F := 𝔽₂) (G := P) Counterexample.val
```

2.4 Extra measures for verification

Essentially the only way in which a formally proved result can be wrong is if the main statement or some definition involved in it is wrong. One can greatly reduce the chance of this happening by proving extra results about the definitions to confirm various properties that are expected to hold. The approach of proving such *test* theorems was also taken in the Liquid Tensor Experiment [Sch22]. Definitions from mathlib have effectively been very well tested by use in many results.

To test the correctness of our definitions of *group rings* and *free modules* – which are relatively complex constructions – we proved a few results that are not needed in the main theorem (some of these were on the suggestion of Giles Gardam).

- The group ring of a group is a ring, in particular the product is associative.
- The inclusion map $G \to R[G]$ given by $g \mapsto g \cdot 1$ is a monoid homomorphism. Further, if R is a field, then the map $g \mapsto g \cdot 1$ is injective.
- The inclusion map $R \to R[G]$ given by $r \mapsto r \cdot 1$ is a ring homomorphism and is injective.

An additional check on the definition of free modules is that the two definitions of quotients on formal sums were proved to be equivalent. Further, we proved a universal property for free modules.

Another test (which we first accidentally carried out) is to slightly vary the formulas for Gardam's units, and check that these are not units.

Our statement involved the definition of *non-trivial* units; while this is a relatively simple definition, it involved our construction of the group ring and not just its structure. To avoid errors arising from this, we proved that our definition was equivalent to the standard one.

Observe that the correctness of our formalisation of a disproof Kaplansky's unit conjecture does not need our group P to coincide with the group which Gardam considered, since it only requires the existence of *a* counter-example. However, we are confident that the counter-example in our code coincides with the one specified by Gardam, since the alternative of finding a different counter-example to the unit conjecture by chance is exceedingly unlikely.

We also show that our construction of metabelian groups satisfies the defining short exact sequence.

2.5 Conclusion

Our formalisation illustrates how softwares such as Lean4 can be used to combine formal proofs with efficient proved algorithms to verify results of the nature of Giles Gardam's theo-

rem in real time. It is one among only a handful of other instances of real-time formalisation, such as the formalization of the Cap set conjecture by Dahmen, Hölzl and Lewis [DHL19], that of the Erdös-Graham density theorem (proved by Thomas Bloom) [Blo21] by Thomas Bloom and Bhavik Mehta and the Liquid Tensor Experiment [Sch22]. We hope that as the size of mathlib and the power of automation tools like `aesop` continue to increase, such formalisations will become more commonplace.

A crucial aspect of the formalisation was the choice of the description of the group P as a metabelian group, which was suitable for both the proof of torsion-freeness and the computation in the group ring. Our heavy use of automation in the form of typeclasses and tactics was another component that made this formalisation fairly quick with a low de Bruijn factor 1.3.3. We hope that a blend of proofs and proved algorithms will be useful for many results.

Chapter 3

The ends of a graph : a formalisation

This chapter describes the concept of the ends of a graph, together with a formalisation of the definition and related concepts in the Lean proof assistant. All work in this chapter is joint with Rémi Bottinelli. Some parts of our code have been integrated with the Lean mathematics library `mathlib`, and some other parts are under review. The main files of the code, including some unfinished portions, have been gathered in the following repository: `https://github.com/0art0/Freudenthal-Hopf`.

The section 3.1 describes the relevant mathematics, including the intuitive idea, a rigorous definition and the notion of functoriality of ends. The next section 3.2 describes parts of our formalisation, with an emphasis on some of our choices of definitions and the insights we had while formalising these concepts.

3.1 The ends of a graph

3.1.1 The idea of ends

The *ends* of a graph (or more generally, a topological space) intuitively represent the distinct directions in which the graph approaches infinity.

Thus, the graph of natural numbers, shown in 3.1.1, has one end which extends to the

right. Similarly, the graph of integers, shown in 3.1.1, has two ends – one extending in each direction.

Figure 3.1: The graph of natural numbers with one end

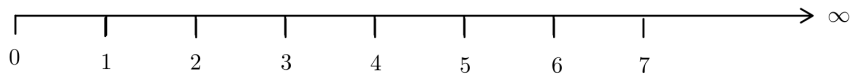
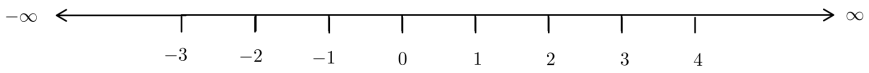


Figure 3.2: The graph of integers with two ends



A more complicated graph such as the one shown in 3.1.1 has infinitely many ends.

A finite graph has no ends, as it does not extend infinitely in any direction.

3.1.2 A rigorous definition of ends

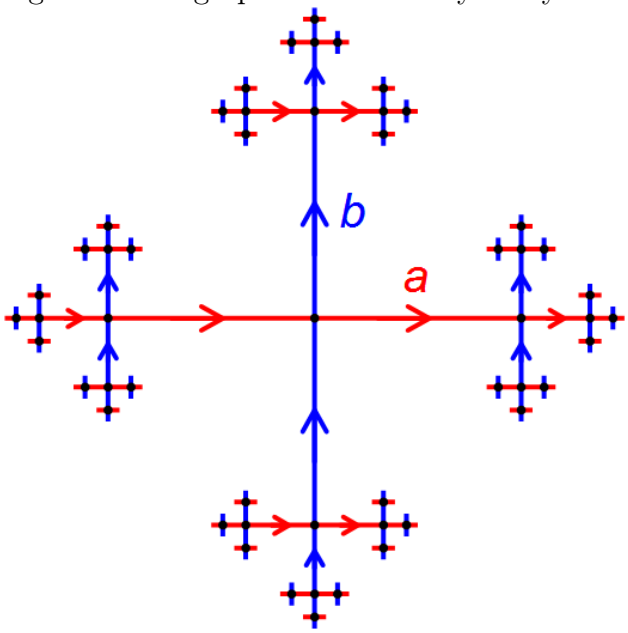
Several definitions have been put forth to make the above intuitive idea of ends precise. We describe one here:

Definition 3.1.1 (Ends)**.** Let G be a graph on a vertex set V. An *end* of G is defined as a function assigning to each finite subset of V a connected component in its complement, subject to a consistency condition that the component assigned to any subset of a finite set K must contain the component assigned to K.

An end therefore indicates a general "direction" of travel outside a chosen finite subset of the graph, with the consistency condition ensuring that this direction is preserved on shrinking or expanding this finite set.

Remark 3.1.2. If K and L are two subsets of the vertex set of a graph with $K \subseteq L$, any non-empty connected component in the complement of L determines a unique connected component in the complement of K that contains it. This follows from the fact that any non-empty connected set of a graph is contained in a unique connected component.

Figure 3.3: A graph with infinitely many ends



Other definitions of ends include the one in terms of rays by Halin [Hal77] and the description as a pursuit-evasion game on a graph (which is similar to the above definition) due to Robertson, Seymour and Thomas [RST91]. These have been proved to be equivalent in [DK03].

The definition 3.1.1 can also be formulated more conceptually in the language of category theory. We recall some relevant category-theoretic notions below.

Definition 3.1.3. A directed set is a set I equipped with a reflexive and transitive binary relation $\leq$ such that for every $a, b \in I$, there is a $c \in I$ such that $a \leq c$ and $b \leq c$.

Remark 3.1.4. A directed set can be regarded as a category whose hom-sets are sub-singletons, equipped with "weak" coproducts.

Definition 3.1.5. Let J be a directed set, regarded as a category in the above sense. Given a category C and a functor $F : J^{\mathrm{op}} \to C$ (known as an *inverse system*), the *inverse limit* of the inverse system is an object X in C equipped with morphisms $\pi_j : X \to F(j)$ for each $j \in J$, such that for any $f : j \to j'$ in J, $\pi_j = F(f) \circ \pi_{j'}$. Moreover, the inverse limit is universal among all such pairs (Y, ϕ_j), in the sense that there must exist a unique morphism $u : Y \to X$ such that $\phi_j = \pi_j \circ u$, for each $j \in J$.

Remark 3.1.6. In the category of sets, and many other concrete categories, the inverse limit

of a system always exists.

Definition 3.1.7. Given a functor $F : C \to \text{Set}$, a *section* of F is a consistent choice of elements $\sigma(x) \in F(x)$ for each $x \in C$, such that for any $f : x \to x'$ in C, $F(f)(\sigma(x)) = \sigma(x')$.

Remark 3.1.8. Each element in the inverse limit of a system $F : J^{\text{op}} \to \text{Set}$ determines a section of F by mapping the chosen element along the projections. Conversely, every section of F determines a unique element of the inverse limit of the system F using the universal property. A section σ can be regarded as a collection of functions $\phi_j : \{\star\} \to F(j)$ from the single-object set; the universal property gives a unique map $u : \{\star\} \to \lim F$ whose image is the required object.

With the above concepts, the definition of ends can be restated as:

Definition 3.1.9 (Ends)**.** Let G be a graph with vertex set V. The set of finite subsets of V form a directed set $\text{Fin}(V)$ under inclusion.

Consider the function $C : \text{Fin}(V) \to \text{Set}$ taking a finite set $K \subseteq V$ to the set $C(K)$ of connected components of the graph $G \backslash K$. From 3.1.2, it follows that if $K \subseteq L$ are finite subsets of V, there is a map from $C(L)$ to $C(K)$. Together, this determines a functor $\text{E} : \text{Fin}(V)^{\text{op}} \to \mathit{Set}$.

The ends are defined to be the sections of the functor E (or equivalently, elements of the inverse limit of E).

Remark 3.1.10. Restricting to the *infinite* connected components outside a given finite set gives the same set of ends.

3.1.3 Functoriality of ends

Sufficiently nice maps between graphs induce maps between the corresponding sets of ends. More concretely, there is a category QGrph whose objects are graphs and a functor End : QGrph $\to$ Set taking a graph to its set of ends. The following heuristic describes the behaviour of the morphisms under the functor:

> Let $\phi : G \to G'$ be a "sufficiently nice" map between graphs, and let e be an end of G. To define an end on G' corresponding to e, we first pick an arbitrary finite

subset L of G'. Now suppose that we are somehow able to produce a finite subset K of G from L; let C denote the connected component in the complement of K determined by the end e. Assuming that ϕ maps C into a connected component in the complement of L, we can construct an end on G' by assigning this component to the chosen set L.

To make this heuristic precise, we require a function assigning to each finite subset L of G' a finite subset K of G, together with the property that the map ϕ sends connected components outside K to connected components outside L. These two conditions abstractly describe maps between graphs which induce maps between the corresponding ends; the next section describes some concrete classes of maps satisfying these properties.

Remark 3.1.11. For a map ϕ between graphs to induce a map of connected components, it is both necessary and sufficient for it to preserve connectivity, i.e., if a, b are vertices in the domain which are connected by a path, their images $\phi(a), \phi(b)$ in the codomain should also be connected by a path.

The induced map between ends is described below in more precise terms. Let G and G' be graphs, and let $\phi : G \to G'$ be a map. Suppose β is a function which takes finite subsets of G' to finite subsets of G, and moreover ensures that for any finite subset L of G, $\beta(L)$ contains $\phi^{-1}(L)$. Then for any such L, it is possible to restrict ϕ to a map ϕ_L from $G \backslash \beta(L)$ to $G' \backslash L$. Suppose that for any L, ϕ_L preserves connectivity, i.e., sends connected components to connected components. Then ϕ induces a map of ends as follows: Let e be an end of G, and let L be a finite subset of G'. The set $\beta(L)$ is a finite subset of G, and the end e determines a connected component C in the complement of $\beta(L)$. Since the map ϕ_L preserves connectivity, mapping along it produces a connected component C_L in the complement of the set L. This determines an end e' on G'. The consistency condition that $e'(L') \subseteq e(L)$ whenever $L \subseteq L'$ can be verified by considering the image of the connected component $e(\beta(L) \cup \beta(L'))$.

3.1.4 Ends of groups

Using the notion of the ends of a graph, it is possible to define the notion of the ends of a *group*. It is possible to pass from groups to graphs via the notion of the *Cayley graph* of a group [Löh17]:

Definition 3.1.12 (Cayley graph)**.** The *Cayley graph* of a group G with generating set S is the graph whose vertices are the elements of G in which all edges are of the form (g, gs) or (gs, g), where $g \in G$ and $s \in S$.

A group can have many generating sets, and thus the definition of a Cayley graph of a group depends on the choice of the generating set. However, it turns out that the Cayley graphs defined by any two generating sets S and S' of a group G are *quasi-isometric*.

Quasi-isometric embeddings and quasi-isometries are "coarse" analogues of isometric embeddings and isometries.

Definition 3.1.13 (Quasi-isometric embedding)**.** Let G and G' be graphs (or more generally, pseudo-metric spaces). A map $\phi : G \to G'$ is a *quasi-isometric embedding* if there exist constants $K \geq 1$ and $C \geq 0$ such that

$$\forall x, y \in G, \frac{1}{K} d_G(x, y) - C \leq d_{G'}(\phi(x), \phi(y)) \leq K d_G(x, y) + C$$

Definition 3.1.14 (Quasi-isometry)**.** A *quasi-isometry* is a map $\phi : G \to G'$ between graphs which is both a quasi-isometric embedding and "coarsely surjective", i.e., every point of G' is within a fixed distance D of $\phi(G)$.

It is also possible to formulate this more conceptually in terms of *coarse bi-Lipschitz maps*. [La 00]

Definition 3.1.15 (Coarse-Lipschitz maps)**.** A map $\phi : G \to G'$ between graphs is *coarse Lipschitz* with constants $K \geq 1$ and $C \geq 0$ if

$$\forall x, y \in G, d_{G'}(\phi(x), \phi(y)) \leq K d_G(x, y) + C$$

Definition 3.1.16 (Coarsely close maps)**.** Two maps $\phi, \psi : G \to G'$ between graphs are considered to be *coarsely close* if

$$\sup_{x \in G} d_{G'}(\phi(x), \psi(x)) < \infty$$

It turns out that a *quasi-isometry* of graphs G and G' can be described more conceptually as a pair of maps $\phi : G \to G'$ and $\psi : G' \to G$ such that $\phi \circ \psi$ and $\psi \circ \phi$ are coarsely close to the identity maps on their respective graphs.

Theorem 3.1.17. *Two maps $\phi, \psi : G \to G'$ between graphs induce the same map between the ends if and only if they are coarsely close (this assumes that the two maps are sufficiently nice to induce maps of ends in the first place).*

Theorem 3.1.18. *The Cayley graphs of a group G defined by two generating set S and S' are quasi-isometric.*

Theorem 3.1.19. *Quasi-isometries satisfy the conditions required for inducing functoriality of ends. In particular, the ends of quasi-isometric graphs are in bijection with each other.*

With these notions, it is possible to define the ends of a group:

Definition 3.1.20. The ends of a group G are defined to be the ends of the Cayley graph for a choice of generating set of G. By the above results, the set of ends of a group is invariant up to bijection under the choice of generating set.

3.2 Formalisation

This section describes the formalisation of *ends* and related properties in `mathlib` by Rémi Bottinelli and the author. We formalise ends in the generality of graphs rather than metric spaces or topological spaces, since it is both more concrete to work with and is sufficient for applications to geometric group theory. Portions of the code forming a part of the background theory are omitted for clarity.

3.3 The definition of ends

While the definition 3.1.1 was initially chosen for concreteness, it turned out to be difficult to use in practice – the fact that an infinite graph has at least one end was in particular quite tedious to prove using this definition. On the suggestion of Kyle Miller, we adopted

the more abstract formulation of ends 3.1.9 in terms of sections of a functor. Because of its category-theoretic nature, it allowed us to prove some properties of ends by specialising general results in `mathlib`'s vast category theory library.

We begin by defining the notion of the connected components in the complement of a given set 3.3.1, and proceed to prove various useful properties about this definition.

```
variables {V : Type u} (G : simple_graph V) (K : set V)

/-- The components outside a given set of vertices ‘K‘ -/
@[reducible] def component_compl := (G.induce K^c).connected_component
```

Listing 3.3.1: The connected components in the complement of a set of vertices.

This definition was chosen after several iterations, taking into account usability for defining ends and compatibility with the foundations of Lean and the conventions of `mathlib`. The cluster of properties surrounding a definition determine the ease with which it can be used in another part of the code. We take care to prove various theorems about disjointness and adjacency of components and vertices that are relevant for our formalisation.

When $K \subseteq L$ are two sets of vertices, any component in the complement of L gives unique component in the complement of K containing it. The listing 3.3.2 contains our definition of this map between components.

```
/--
If ‘K ⊆ L‘, the components outside of ‘L‘ are all contained in a single component
    outside of ‘K‘.
-/
@[reducible] def hom (h : K ⊆ L) (C : G.component_compl L) : G.component_compl K
    :=
C.map (induce_hom hom.id (set.compl_subset_compl.2 h))
```

Listing 3.3.2: The map between connected components in the complement

Finally, we define the functor sending finite sets of vertices to the set of connected components in their complement (as described in 3.1.9) and define the ends of a graph as the sections of this functor 3.3.3.

```
/--
```

```
The functor assigning, to a finite set in ‘V‘, the set of connected components in
    its complement.
-/
@[simps] def component_compl_functor : (finset V)^op => Type u :=
{ obj := λ K, G.component_compl K.unop,
  map := λ _ _ f, component_compl.hom (le_of_op_hom f),
  map_id' := λ K, funext (λ C, C.hom_refl),
  map_comp' := λ K L M h h', funext (λ C, C.hom_trans (le_of_op_hom h)
    (le_of_op_hom h')) }

/-- The end of a graph, defined as the sections of the functor
    ‘component_compl_functor‘ . -/
@[protected]
def «»end := (component_compl_functor G).sections
```

Listing 3.3.3: The definition of the ends of a graph

With this definition of ends, we can prove that an infinite locally-finite graph has at least one end by specialising an abstract result from category theory which states that the inverse limit of an inverse system of non-empty, finite sets is non-empty.

```
/--
A locally finite preconnected infinite graph has at least one end.
-/
lemma nonempty_ends_of_infinite [Glf : locally_finite G] [fact $ preconnected G]
    [Vi : infinite V] :
  G.end.nonempty :=
by classical; apply nonempty_sections_of_finite_inverse_system
    G.component_compl_functor
```

Listing 3.3.4: An infinite locally-finite graph has at least one end

3.4 The functoriality of ends

We define functoriality of ends with two goals in mind:

- Abstractly describing the maps between graphs which induce maps between the corresponding ends in the greatest possible generality.
- Proving that a concrete class of maps – namely the quasi-isometric embeddings (or coarse Lipschitz maps with coarse Lipschitz inverses) – induce functoriality of ends.

We recall the abstract characterisation of maps that induce functoriality of ends, as described in 3.1.3. A map $\phi : G \to G'$ between graphs induces functoriality of ends if

- There is a function β is a function which takes finite subsets of G' to finite subsets of G, such that for any finite subset L of G', $\beta(L)$ contains $\phi^{-1}(L)$.
- The restriction of the map ϕ to the domain $G \backslash \beta(L)$ and range $G' \backslash L$ preserves continuity, in the sense that it sends connected sets to connected sets.

While formalising this in Lean, we realised that the second condition on preserving continuity can be expressed more succinctly by defining the notion of an ∞-*Lipschitz map* (i.e., a coarse Lipschitz map where the scaling constant takes the value ∞). This rests on the observation that two points of a graph are connected by a path if and only if their distance is strictly less than infinity. We modify the definition of a coarse Lipschitz map to an equivalent one that accommodates the case where the scaling constant K takes the value ∞.

Definition 3.4.1 (Coarse Lipschitz map)**.** A map $\phi : G \to G'$ between graphs is *coarse Lipschitz* with constants $K \geq 1$ and $C \geq 0$ if

$$\forall x, y \in G, \forall a, d_G(x, y) < a \implies d_{G'}(\phi(x), \phi(y)) < K \cdot a + C$$

In code, this is formulated as

```
def coarse_lipschitz_with (K : ℕ∞) (C : ℕ) (f : V → V') :=
  ∀ {⦃x y : V⦄}, ∀ {⦃a⦄}, G.edist x y < a → G'.edist (f x) (f y) < K * a + C
```

Listing 3.4.1: The definition of a coarse Lipschitz map

Equipped with this definition, we define a structure capturing the properties required for a function to induce a map between ends.

```
/-- The kind of map between graphs which induces a map on the ends. -/
structure coarse_map {V V' : Type u} (G : simple_graph V) (G' : simple_graph V')
    (f : V → V') :=
  (K : ℕ∞) (C : ℕ)
  (finset_mapping : finset V' → finset V)
  (finset_inv_sub : ∀ L : finset V', f ⁻¹' (L : set V') ⊆ (finset_mapping L : set
   V))
  (induced_coarse_lipschitz : ∀ L : finset V',
    coarse_lipschitz_with (G.induce (finset_mapping L)ᶜ) (G'.induce Lᶜ)
      K C (induce_out f (finset_inv_sub L)))
```

Finally we use this definition to prove the functoriality of ends.

```
def coarse_map.end_map [decidable_eq V] {f : V → V'} (fcoarse : coarse_map G G'
  f) : G.end → G'.end := by
{ ... }
```

Listing 3.4.2: A proof that coarse maps induce maps of ends

3.5 Conclusion

This chapter describes work (done in collaboration with Rémi Bottinelli) towards formalising the concept of the *ends* of a graph in Lean's mathematics library `mathlib`. The formulation of the definitions and concepts required several iterations to ensure compatibility with the relevant library definitions while also maintaining their usability. By stating some definitions in more generality, we were able to specialise existing library results to obtain short and conceptual proofs of otherwise tedious theorems. The formalisation also led to some new insights, albeit small, such as the notion of ∞-Lipschitz maps. Future work in this direction could include formalising well-known theorems about the ends of graphs, such as the *Freudenthal-Hopf theorem* or *Stallings' theorem about ends*, to test the usability of this definition of ends.

Chapter 4

A formalisation blueprint for Stallings' topological proof of Grushko's theorem

4.1 Introduction

Grushko's theorem [Gru40] is a group-theoretic result concerning the ranks of finitely-generated groups. For finitely-generated groups A and B, Grushko's theorem states that

$$\operatorname{rank}(A \star B) = \operatorname{rank}(A) + \operatorname{rank}(B)$$

where $A \star B$ denotes the free product of A and B and the *rank* of a group refers to the smallest cardinality of a generating set of that group. In other words, Grushko's theorem states that the rank of a free product of finitely-generated groups is equal to the sum of the ranks of the individual groups. The bound $\operatorname{rank}(A \star B) \leq \operatorname{rank}(A) + \operatorname{rank}(B)$ follows from the structure of the free product, since any generating sets X of A and Y of B give a generating set $X \cup Y$ of $A \star B$; Grushko's theorem shows that this inequality is in fact an equality.

Grushko's theorem can be easily deduced from a more general lemma about free groups and free products:

Lemma 4.1.1 (Grushko)**.** *Consider a finitely-generated free group F and two finitely-generated*

groups A and B, and let $\phi : F \to A \star B$ be a surjective *homomorphism. Then ϕ splits, i.e., there are subgroups F_1 and F_2 of F such that $F = F_1 \star F_2$ and $\phi(F_1) = A$ and $\phi(F_2) = B$.*

Grushko's theorem follows from 4.1.1 by viewing a finitely-generated group as one *marked* by a surjective homomorphism from a finitely-generated free group and using the fact that any sub-group of a free group is free (a well-known result known as the *Nielsen-Schreier theorem* [Nie21] [Sch27]).

This result has been studied and generalised in various contexts. The lemma 4.1.1 can be extended to involve the free product of arbitrarily many groups. The lemma 4.1.1 has also been studied and substantially generalised by Higgins using groupoid techniques [Hig66]. Notably, the purely-group theoretic result 4.1.1 has been given an elegant topological proof by Stallings [Sta65]; this proof is a remarkable instance of topological methods being used in group theory [SW79], and is the focus of this chapter.

Stallings' topological proof is presented in detail in the next section 4.2, adapted primarily from [SW79], but with elements of Stallings' original proof [Sta65]. The bridge between the worlds of topology and group theory in Stallings' proof comes from the fundamental group. Though the result can be stated and presented for general topological spaces, it is better understood using a combinatorial model of space in the form of CW complexes. It suffices to consider finite two-dimensional CW complexes – which consist of disks bounding loops on an underlying graph – as every finitely-presented group is the fundamental group of some finite two complex and the fundamental group of a CW complex depends only on its two-skeleton. Thus, paths become edge paths, generators and group elements become edge loops at a fixed base-point on the two-complex, and relations become disks on the two-complex. Stallings' proof crucially relies on the idea of a *binding tie* (defined in 4.2), which makes it possible to iteratively reduce the number of connected components in a certain sub-complex until there is only one component. When there are finitely many groups involved, Stallings' proof is algorithmic and constructive, and can be used to explicitly identify the subgroups in the splitting of the free group in 4.1.1.

Stallings' topological proof of Grushko's theorem is an interesting target for formalisation for various reasons. Grushko's theorem is an important mathematical result that has not been formalised in any proof assistant so far (to the best of our knowledge). Stallings' proof involves a delicate interplay of algebra, topology, visual intuition, and algorithms.

Formalising the result and structuring the definitions and ideas in a way that captures both the topological and algorithmic aspects is a challenging and worthwhile endeavour. A constructive formalisation of the result will not only certify that the result is correct, but also give a way to compute the generators of the two subgroups of the free group in 4.1.1. Thus it may be regarded as an algorithm with proof for computing the splitting in 4.1.1. Moreover, such a formalisation can be combined with graph rendering tools to produce visualisations accompanying the execution of the binding tie algorithm, which may aid in understanding the proof. A formalisation that combines mathematical proofs with algorithms and visualisation may serve as inspiration for other formalisations of a broadly similar nature.

This chapter presents Stallings' proof of Grushko's theorem in 4.2. The rest of the chapter is devoted to describing a possible formalisation *blueprint* – a detailed description of the proof and the definitions involved in a way that is suitable for formalisation. Care is taken to ensure that the definitions can be used for computation, and also that the computations are reasonably efficient. The binding tie algorithm is described keeping in mind the eventual goal of exporting the graph structure of the complex to an external tool for visualisation. Formalisation of this proof in the Lean4 proof assistant is not yet complete at the time of writing.

4.2 Stallings' topological proof of Grushko's theorem

This section describes a sketch of the proof of 4.1.1 due to Stallings, which uses topological ideas to establish this group-theoretic result. Finitely-presented groups are viewed as the fundamental groups of finite two-complexes, which opens up the problem to topological methods. The proof in this section follows the standard notation and terminology of the algebraic topology and graph theory literature.

To prove the result, consider two pointed CW complexes (C, c) and (D, d) with fundamental groups isomorphic to A and B respectively. The wedge product of C and D with their respective base-points produces a pointed CW complex $(C \vee D, v)$ with fundamental group isomorphic to the free product $A \star B$. Consider a pointed CW complex (X, x) whose fundamental group is the free group F, together with a surjective base-point preserving continuous map $f : (X, x) \to (C \vee D, v)$ such that $\pi_1(f) = \phi$. This is the basic topological set-up corresponding to the group-theoretic statement.

It is convenient to initially choose X to be a *graph* (i.e., a two-complex without disks), such as a bouquet of circles. Further, it may be assumed that X contains no loops with null-homotopic images under f (also called *null loops*). A null loop in a graph implies the existence of a generator of F (the fundamental group of (X, x)) which maps to the identity element of $A \star B$ under ϕ; such generators can be temporarily excluded from a given generating set and included back in either component of the splitting of the resulting subgroup.

Let Y be the subgraph of X corresponding to the pre-image of the base-point v of $C \vee D$ (i.e., the unique point in $C \cap D$). By the arguments of the preceding paragraph, the complex X can always be chosen so that this subgraph is a forest. The subgraph Y is the intersection of the subgraphs $f^{-1}(C)$ and $f^{-1}(D)$ – the pre-images of the two components of the wedge product.

In the special case where this forest is in fact a tree, the result admits an easy proof. The complex X can be written as the union of $f^{-1}(C)$ and $f^{-1}(D)$, which intersect in a tree – a simply-connected space. By Van Kampen's theorem, $\pi_1(X, x) = \pi_1(f^{-1}(C), x) \star \pi_1(f^{-1}(D), x)$. Since $\pi_1(X, x) = F$, $\pi_1(f^{-1}(C), x)$ and $\pi_1(f^{-1}(D), x)$ are the two subgroups required for the splitting. By construction, $\phi(\pi_1(f^{-1}(C), x)) \subseteq A$ and $\phi(\pi_1(f^{-1}(D), x)) \subseteq B$; the surjectivity of ϕ and with properties of the free product of groups ensure that these containments are in fact equalities. The rest of the proof involves reducing the general case to the one where the forest is a tree.

The general proof rests on the notion of a *binding tie*, which is a special kind of path p in X satisfying the following properties:

1. *Monochromatic:* $f(p) \subseteq C$ or $f(p) \subseteq D$.
2. *Tie:* The endpoints of p lie in different components of the forest Y.
3. *Null:* The loop $f(p)$ is homotopic to the null loop at v, the base-point of $(C \vee D, v)$.

Given a binding tie, it is possible to transform X to a different complex X' with the same fundamental group but strictly fewer connected components in its forest, such that the set-up of the problem is preserved.

This transformation can be thought of as a *Whitehead move*, which was defined by Whitehead in the context of simple homotopy theory [Whi50]. Given a path p in X between points

a and b, the modification consists of adding an extra edge e connecting a and b, together with a disk bounding the loop $p \cdot e^{-1}$. Since the modified space deformation retracts to the original space X, it has the same fundamental group F. When this path p is a binding tie, it is further possible to extend the map f to the modified space such that the set-up is preserved. The image of a binding tie under the map f is a loop based at v that is homotopic to the null loop at v. By choosing the image of the edge e to be the null loop at v and the image of the disk bounding $p \cdot e^{-1}$ to be the null-homotopy of $f(p)$, it is possible to extend f in a way that preserves the original map of fundamental groups ϕ. Since a binding tie connects different components of the forest Y, this construction also ensures that the forest in the modified complex has strictly fewer connected components.

As long as the existence of binding ties is guaranteed, it is possible to carry out the above construction iteratively until the problem is reduced to the base case where the forest in the pre-image is in fact a tree. We now show the existence of binding ties when the forest Y contains more than one component.

Since the original complex X is connected, it is possible to find a path p in X connecting two points from distinct components of the sub-complex Y; for convenience, we choose one of these points to be the basepoint x of the complex X. The path p connects distinct components in Y, but is neither monochromatic nor is its image null-homotopic; the strategy will be to gradually make modifications until a path satisfying all three conditions for being a binding tie is found. Since both endpoints of p are in Y, its image in under f is a loop at v, i.e, an element in the fundamental group $\pi_1(C \vee D, v) = A \star B$. Since ϕ is a surjective homomorphism of groups, it is possible to find a loop q in X based at x which such that its image $f(q)$ lies in the same homotopy class as $f(p)$. Now the modified path $q^{-1} \cdot p$ is also a path connecting distinct components of Y (in fact, the same components that p was connecting earlier) whose image under f is *null-homotopic* by construction. Thus two of the three conditions are now satisfied; it only remains to find a monochromatic path with these two properties. The path $q^{-1} \cdot p$ can be divided into a finite number of *maximal monochromatic segments* $l_1 \cdot l_2 \ldots l_n$, such that any two adjacent segments are of opposite colours. In particular, this implies that the end-points of each segment l_i lie in Y. Mapping these under f, we obtain a collection of loops based at v whose product in the correct sequence is homotopic to the null loop at v. In the fundamental group $A \star B$, this translates to a finite set of elements $\{a_1, a_2, \ldots, a_n\}$ whose product is the identity element. By a general theorem about free products, it turns out that if $a_1 \cdot a_2 \cdot \ldots \cdot a_n = 1$, then $a_i = 1$ for some

$1 \leq i \leq n$. This implies that one of the monochromatic segment l_i is in fact null-homotopic. If the end-points of l_i lie in different components of Y, the construction is complete and l_i is a binding tie. Otherwise, we can replace l_i with a path between its endpoints that is completely contained inside Y and repeat the process by including the new path with l_{i-1} and l_{i+1}. Since the number of monochromatic segments in the modified path is strictly fewer, the process eventually terminates to produce a binding tie.

4.3 A formal blueprint

This blueprint presents Grushko's theorem taking into account inter-operability of definitions, compatibility of algebraic, topological and algorithmic notions in the proof, and suitability for computation and visualisation.

4.3.1 Graph algorithms with proof

Stallings' proof is effective, which makes it possible to implement the binding tie proof in a way that can be run on inputs by selecting suitable data structures for efficient computation.

Most of the proof involves manipulating the underlying 1-complex of the space X, making it necessary to have good descriptions and data-structures for 1-complexes or graphs that can be used in both proofs and algorithms.

Graphs

Graphs (more precisely, symmetric multigraphs) can be conveniently described in the foundations of Lean by adapting a definition originally due to Serre.

A *graph* consists of a set of vertices V, a set of arrows $\mathrm{Hom}(v, w)$ for every pair of vertices $v, w \in V$ and a family of maps $\mathrm{inv}_{v,w} : \mathrm{Hom}(v, w) \to \mathrm{Hom}(w, v)$ satisfying the following conditions:

- $\forall e \in \mathrm{Hom}(v, w), \mathrm{inv}_{w,v}(\mathrm{inv}_{v,w}(e)) = e$

- $\forall e \in \mathrm{Hom}(v, v), \mathrm{inv}_{v,v}(e) \neq e$

Thus, a symmetric graph is represented as a directed graph with twice the number of edges – one representing the edge and the other representing its inverse.

The paths on a graph can be defined inductively: there is a *null path* nil_v from any vertex to itself, and given $e \in \mathrm{Hom}(u, v)$ and a path p from v to w, we have a path $\mathrm{cons}(e, p)$ from u to v. The concatenation of paths can also be defined inductively.

We would like to impose the following conditions on paths to allow *edge cancellation*:

- $\forall e \in \mathrm{Hom}(v, w), e \cdot \mathrm{inv}_{v,w}(e) = \mathrm{nil}_v$
- $\forall e \in \mathrm{Hom}(v, w), \mathrm{inv}_{v,w}(e) \cdot e = \mathrm{nil}_w$

Quotienting by these relations gives paths up to edge cancellation, which are elements of the fundamental groupoid of the graph; focusing on loops based at a chosen vertex gives the fundamental group.

4.3.2 Listable types

Finiteness is a recurring theme in Stallings' proof. The spaces under consideration are all finite, the groups are finitely-presented and the binding tie algorithm crucially uses the finiteness of the size of the forest to ensure termination.

The definition of a graph presented above is a general one, which works even for graphs with infinite vertices.

To effectively handle finiteness, we can define a type to be `Listable` if there is a list which contains all its elements without having any duplicates (lists are data structures defined in Lean, and are always finite by construction). Thus a finite graph will be one with a `Listable` vertex set and `Listable` Hom-sets. A `Listable` type also has an implicit order – the order in which the elements occur in the list – and this allows `Listable` types to be enumerated for programming-related applications. To make the `Listable` types usable in practice, it is necessary to also prove several theorems that capture our basic intuitions about finiteness.

4.3.3 Depth-first search

Several steps of the proof require concrete computations with graphs to test various properties such as connectedness and to find spanning trees of graphs. The depth first search algorithm, which is a standard graph algorithm, is the ideal choice for carrying out these various tasks.

The basic idea behind the depth-first search algorithm is intuitive and can be described concisely: We start with a finite graph G and a chosen vertex v. We also maintain a list of visited vertices, which is initially set to the empty list. Adding v to the list of visited vertices, we look at its list of neighbours in the graph. For each neighbour in the graph that has not already been visited, we recursively execute the depth-first search algorithm at that vertex using the updated list of visited vertices.

The output of a depth-first search contains useful information such as the set of vertices reachable from the initial vertex v and a tree that spans this set of vertices.

A proved implementation of the depth-first search algorithm can be used for various tasks, including:

- Checking whether two vertices in a graph are connected by a path
- Finding the connected components of a graph
- Finding a spanning tree for a graph

All three occur at various points in the proof, and capturing them with a single algorithm is useful. Finding a spanning tree efficiently is especially relevant in order to compute a set of generators for the fundamental group of a graph.

The depth-first search algorithm described above is a classic example of an *imperative program*, in the style of languages such as `Python` and `C++`; however, `Lean` is a functional programming language, in the tradition of `Haskell` and `OCaml`. A notable formalisation of a graph search algorithm due to Natarajan Shankar in [Sha10] circumvents this difficulty by formulating a general search algorithm in the language of fixed points and lattices. However, it is unclear whether this idea can be adapted to the case of depth-first search since the set of sub-trees of a given graph do not form a complete lattice.

4.3.4 Graph-marked groups

It is customary to extend the previously defined notion of the fundamental group of a graph to the setting of two-complexes. However, despite its conceptual convenience, this approach is difficult to work with computationally.

Just as the definition of a graph requires edges to be present in pairs, the definition of a two-complex requires several copies of a two-cell, one for each point on the boundary. The resulting symmetry conditions can be difficult to verify while constructing two-complexes in practice.

A more serious drawback is that the fundamental group of a two-complex is not described abstractly, but rather in terms of generators and relations. A basic requirement for working with a group computationally is to have a way of deciding when two elements of the group are equal. With a description in terms of generators and relations, this requires solving the word problem for the given group. This requires leveraging properties of the specific group, because it is known to be impossible to find a solution to the uniform word problem; moreover, even if the word problem can be solved, it is not guaranteed to be efficient.

We therefore deviate from the standard approach and define the notion of a *graph-marked group* to introduce non-trivial homotopies of paths.

A *graph-marked group* consists of a pointed graph (Γ, v) and a group G together with a *surjective* homomorphism from $\pi_1(\Gamma, v)$ to G.

This surjection is also required to be effective, meaning that it should be possible to produce an explicit loop in the pre-image of each element of G.

Two loops p, q based at v are considered *homotopic* if $\phi([p]) = \phi([q])$. The fundamental group of a graph-marked group is *defined to be* the group G. This is the same result that we would get by quotienting the loops of the graph by the equivalence relation introduced by homotopies.

This allows the group to be an explicit part of the data, rather than something to be computed from the combinatorial representation of the topological space. In particular, this cleanly avoids the several layers of isomorphisms of groups that would otherwise arise while executing the binding tie algorithm. The notion of a graph-marked group allows one to speak

about homotopy of loops without recourse to two-complexes and generators and relations. While this replaces the elegant topological picture of two-complexes, it introduces a structure connecting the topological and algebraic pictures in a way that is suited for the proof.

Homotopy of paths in a graph-marked group

Note that the notion of homotopy of paths can be extended to arbitrary pairs of paths with the same endpoints. Suppose $x, y \in V$ and p, q are paths from x to y. Since the graph is connected we can find paths r, s from the basepoint v to x and y respectively and check whether the loops $r \cdot p \cdot s^{-1}$ and $r \cdot p \cdot s^{-1}$ are homotopic in the sense of having the same image under the homomorphism ϕ. This is well-defined, since if r' and s' are a different set of paths from v to x and y respectively, then

$$\begin{aligned}
\phi(r' \cdot p \cdot s'^{-1}) &= \phi(r' \cdot r^{-1} \cdot r \cdot p \cdot s^{-1} \cdot s \cdot s'^{-1}) \\
&= \phi(r^{-1} \cdot r) \cdot \phi(r \cdot p \cdot s^{-1}) \cdot \phi(s \cdot s^{-1}) \\
&= \phi(r^{-1} \cdot r) \cdot \phi(r \cdot q \cdot s^{-1}) \cdot \phi(s \cdot s^{-1}) \\
&= \phi(r' \cdot r^{-1} \cdot r \cdot q \cdot s^{-1} \cdot s \cdot s'^{-1}) \\
&= \phi(r' \cdot q \cdot s^{-1})
\end{aligned}$$

using the fact that ϕ – being a group homomorphism – distributes over path homotopy classes of loops.

Whitehead moves on a graph-marked group

A key part of Stallings' proof involves modifying the two-complex by a Whitehead move in a way that preserves the fundamental group. This construction can be adapted to the setting of graph-marked groups.

Let $((\Gamma, v), G, \phi : \pi_1(\Gamma, v) \to G)$ be a graph-marked group, and consider a path p on Γ from x to y. We first modify the graph Γ by adding an extra edge e from x to y. For every path in the new graph, we can obtain a corresponding path in the original graph by replacing

all occurrences of the edge e with the path p. The homomorphism ϕ can thus be extending to the new graph by first transforming a given path homotopy class to one in the original graph and then mapping along ϕ. The fundamental group of the new graph-marked group is the same as the original one.

Morphisms of graph-marked groups

Given two graph-marked groups $((\Gamma, v), G, \phi)$ and $((\Gamma', v'), G', \phi')$, a morphism between them consists of a pair of maps – a base-point preserving graph homomorphism $\psi : (\Gamma, v) \to (\Gamma', v')$, and a group homomorphism $\rho : G \to G'$, such that $\rho \circ \phi = \phi' \circ \psi$.

4.3.5 The proof

Ground work

The first step in proving Grushko's theorem is to lift the algebraic set-up into the topological world. As input, we are given finitely-presented groups A and B, together with an *effective surjection* (i.e., a surjection together with a means of computing an element in the pre-image of each point in the co-domain) from a finitely-generated free group F to the free product $A \star B$. As described already, it can be assumed without loss of generality that the free group has no generators mapping to the identity element of the free product.

To construct the topological picture, we first create two graphs-marked groups C and D for A and B respectively, each containing a single point and one loop for each generators (i.e., a bouquet of circles). We then take the wedge product of the graphs using their respective base-points to obtain a graph-marked group for the free product $A \star B$.

To construct a graph-marked group X for the free group F, we do not represent each generator by just a single edge in a bouquet of circles, but instead segment each such loop into multiple segments such that the number of segments corresponds to the length of the path representing the image of the generator. This naturally gives a graph homomorphism f that represents the original group homomorphism ϕ. By construction, all graphs involved are connected, and the pre-image of the base-point of $(C \vee D, v)$ is a disjoint set of points –

therefore a forest.

4.3.6 The binding tie algorithm

Before running the binding tie algorithm, we first modify the complex $(C \vee D, v)$ with a Whitehead construction applied to the trivial path nil_v. This turns out to be necessary to extend the homomorphism f at each stage of the construction; the new edge added to X gets mapped to the edge added by the above Whitehead construction.

Given a binding tie on X, we can first modify X with a Whitehead construction on the binding tie and then extend the map f by sending the extra edge to the loop created at the trivial path nil_v by the Whitehead construction.

This creates a new set-up of graph-marked groups which is equivalent to the previous one.

The number of connected components in the pre-image of the base-point decreases, which ensures that the process eventually terminates. This can be proved by showing that the connectivity relation on the new graph strictly contains the connectivity relation on the previous graph since an extra connection is made.

The existence of binding ties

We can pick a path between two distinct components of the forest (the pre-image of the basepoint of the co-domain) by first computing the connected component of the basepoint of (X, x) and then picking a point of the forest in its complement, if one exists (otherwise, we proceed to the base case). Since the space X is connected, it is possible to find a path p between the base-point and the chosen point. By computing the image of p and using effective surjectivity, it is possible to recover a path q based at x with the same image. We set $r := q^{-1}p$, which is a null-homotopic path connecting different components of the forest.

The proof of existence of binding ties requires a way of decomposing the path r into maximal monochromatic segments. This can be done algorithmically starting from the left of the path and working towards the end, storing the alternating path in a custom data-

structure.

We map the segments along f to obtain a list of loops whose product is the identity element of $A \star B$. The Lean mathematics library `mathlib` contains the result needed to conclude that one of these elements must be the identity. Following the argument from earlier, we can identify a monochromatic segment in the source which is null-homotopic. If it does not connect distinct components of the forest, we can replace it with a path in the forest and proceed recursively until a binding tie is found.

The base case

We finally come to the case where the forest is in fact a tree. The complex X has been modified at each stage of the algorithm by adding edges through the Whitehead construction. It however turns out that all these edges lie in the tree, since binding ties are always null-homotopic and the extra edges map to the intersection of the complexes C and D in the co-domain.

Finishing the result appears to require a combinatorial version of the general Van-Kampen theorem for graph-marked groups, though it may be possible to perform an elementary construction that avoids this (one such attempt was made by the author).

On deducing that $\pi_1(X, x) = \pi_1(f^{-1}(C), x) \star \pi_1(f^{-1}(D), x)$, the result follows. Labelling $\pi_1(f^{-1}(C), x) = F_1$ and $\pi_1(f^{-1}(D), x) = F_2$, we see that $\phi(F_1 \star F_2) = A \star B$. By construction, $\phi(F_1) \subseteq A$ and $\phi(F_2)) \subseteq B$. That these containments are in fact equalities can be deduced from the surjectivity of ϕ and properties of the free product of groups.

4.4 Visualisation

Stallings' proof, especially the binding tie algorithm, naturally lends itself to visualisation. The binding tie algorithm can be regarded as a sequence of modifications taking place in the source space representing the free group. By capturing a combinatorial description of this space at each step of the process and exporting to an external tool, it may be possible to visualise the execution of the binding tie algorithm on specific inputs. Moreover, such a

visualisation can be played directly in the Lean editor by means of a feature known as user widgets.

The view-point of graph-marked groups has two advantages over the usual approach using two-complexes when it comes to visualisation:

- Graph-marked groups are more minimal than two-complexes, in the sense that they involve only vertices and edges, rather than vertices, edges and two-cells. With fewer details to plot, visualisation using graph-marked groups may be less cluttered.
- While there are many softwares for visualising graphs, there are not as many for visualising two-complexes. This makes it practically easier to create visualisations of the binding tie algorithm using only graphs.

Since the edges of a graph formally occur in pairs, care will have to be taken while exporting to avoid duplication.

4.5 Conclusion

This chapter presents Stallings' topological proof of Grushko's theorem together with a blueprint outlining a possible route for formalisation. The blueprint attempts to connect the various parts of the proof to describe not just a sketch of a formal proof but also a means of computing the generators and visualising the binding-tie algorithm.

The viewpoint of *graph-marked groups* seems to be well-suited for the proof and clears some of the obstacles in the path of formalising the result. Some aspects of the blueprint may need further modifications or refinements; these details may be clearer after some portions of the proof have been formalised. One potential challenge that remains is dealing with finiteness in the various forms in which it occurs in the proof. It is hoped that this blueprint can serve as a potential starting point for formalising Stallings' topological proof of Grushko's theorem.

Chapter 5

Solving equations in Abelian groups

This chapter describes a method of deciding whether a given equation (involving only addition, subtraction and negation operations) holds in all Abelian groups. The idea and the implementation described here are original. Given its elementary nature, it is possible that this idea could have occurred to others before.

5.1 A general equality problem for Abelian groups

Consider the problem of deciding whether of deciding whether an equation, such as

$$(x + y) + z + -(x + z) = y$$

is true in all Abelian groups. This means that for any Abelian group A, and for any chosen values x, y and z in A, the equation should be true.

Humans can solve an arbitrary instance of this problem with ease by "expanding", "cancelling", "rearranging" and "grouping" the terms involved.

However, this task is not trivial to carry out on a computer if one requires a full and rigorous proof in the end. The problem of deciding whether an equation holds in all Abelian groups is the same as deciding whether the equation can be deduced from the axioms for

Abelian groups, and it is in the latter form that the task is usually presented to the computer.

The computer sees the equation not as a single line that can be read from left to right, but rather as an equality of two complicated nested expression trees involving addition, subtraction or negation operators at the nodes and variables at the leaves. This means that the computer cannot ignore the parentheses and tinker easily with the order of the variables the way humans do. A fully-bracketed version of the above looks like

$$(((x) + (y)) + (z)) + (-((x) + (z))) = (y)$$

Any proof from the axioms becomes a complicated ordeal of shuffling parentheses and swapping the order of adjacent variables, and even cancelling adjacent terms in an expression is a challenge when they are not immediately neighbours in the expression tree. Moreover, even if one is able to produce an algorithm that proves these equalities directly from the axioms, it is unlikely that the algorithm will have a good running time (in terms of the number of variables in the expression, including duplicates).

5.2 A solution to the general equality problem for Abelian groups

It turns out that to prove that a formula involving n variables is true in all Abelian groups, it suffices to show it for a specific set of n elements in a specific Abelian group, namely the basis of the free Abelian group on n elements ($\mathbb{Z}^n$).

The crucial property relevant here is that $\mathbb{Z}^n$ is a free Abelian group, which by definition means that any map from the basis $\{(0, \ldots, 0, i, 0, \ldots, 0) \mid 1 \leq i \leq n\}$ to an Abelian group A extends uniquely to an Abelian group homomorphism from $\mathbb{Z}^n$ to A.

5.2.1 A worked example

The general solution is perhaps best illustrated by working out a specific example such as the one above:

Plugging in $(1,0,0),(0,1,0),(0,0,1)$ for x,y,z in the above example yields the equation

$$((1,0,0)+(0,1,0))+(0,0,1)-((1,0,0)+(0,0,1))=(0,1,0)$$

which is true since both sides compute to $(0,1,0)$.

Now consider an arbitrary Abelian group A, and elements a,b,c in A. The function taking the basis element $(1,0,0)$ to a, $(0,1,0)$ to b, $(0,0,1)$ to c extends to a unique homomorphism $\phi:\mathbb{Z}^n\to A$, and evaluating ϕ on the two sides of the previous equation gives

$$\phi\left(((1,0,0)+(0,1,0))+(0,0,1)-((1,0,0)+(0,0,1))\right)=\phi\left((0,1,0)\right)$$

Distributing the homomorphism across addition, subtraction and negation turns the expression into

$$(\phi((1,0,0))+\phi((0,1,0)))+\phi((0,0,1))-(\phi((1,0,0))+\phi((0,0,1)))=\phi((0,1,0))$$

which is precisely the required equation

$$(a+b)+c-(a+c)=b$$

5.2.2 The general solution

The solution in the worked example can be directly generalised to work for any equation.

For an equation involving n variables, consider the free Abelian group on n generators. Verifying the equation for the n generators is a matter of direct computation. This is enough to prove it for all Abelian groups, since for any Abelian group A and n elements $a_1, a_2, \ldots, a_n$ in A, there is a homomorphism sending the generators of the free group to these elements, and any equation involving only the Abelian group operations (addition, negation, subtraction) is preserved under homomorphisms.

Remark 5.2.1. In a way, proving an equality by passing to the free group makes precise the intuitive operations of rearranging and grouping mentioned earlier. The order on the basis elements of the free group fixes an implicit order on the variables in the equation, and the normal form imposed by the tuples of integers handles grouping and cancellation of variables.

5.3 Implementing the solution

This section describes our attempt at implementing the above idea as a `Lean4` meta-program.

We first define a type `AddTree` α which captures the *syntax* of expressions involving addition, negation and subtraction operations performed on variables of type α 5.3.1. This process of passing from an expression in the language to a concrete term that represents it is usually known as *reflection* or *internalisation*. The original expression can be recovered from the `AddTree` term representing it using the function `fold` defined in 5.3.2.

```
inductive AddTree (α : Type _) where
  | leaf : α → AddTree α                          -- variables
  | negLeaf : AddTree α → AddTree α               -- unary negation
  | node : AddTree α → AddTree α → AddTree α      -- binary addition
  | subNode: AddTree α → AddTree α → AddTree α    -- binary subtraction
```

Listing 5.3.1: The `AddTree` inductive type which captures the syntax of expressions involving Abelian group operations

```
def AddTree.fold {α : Type u} [AddCommGroup α] : AddTree α → α
  | AddTree.leaf a => a
  | AddTree.negLeaf t => -(fold t)
  | AddTree.node l r =>  (fold l) + (fold r)
  | AddTree.subNode l r => (fold l) - (fold r)
```

Listing 5.3.2: Interpreting an `AddTree` term back as an expression in the Abelian group

Given a function `f : α → β` between types, there is an induced map between the corresponding `AddTree`s 5.3.3. This is later used in mapping from the free Abelian group back to any chosen group.

```
def AddTree.map {α β : Type _} (f : α → β) : AddTree α → AddTree β
  | AddTree.leaf a => AddTree.leaf (f a)
  | AddTree.negLeaf a => AddTree.negLeaf (map f a)
  | AddTree.node l r => AddTree.node (map f l) (map f r)
  | AddTree.subNode l r => AddTree.subNode (map f l) (map f r)
```

Listing 5.3.3: The induced map on `AddTree`s

```
def AddTree.reduce {α : Type _} : AddTree (AddTree α) → AddTree α
  | AddTree.leaf adt => adt
  | AddTree.negLeaf adt => AddTree.negLeaf (reduce adt)
  | AddTree.node lt rt => AddTree.node (reduce lt) (reduce rt)
  | AddTree.subNode lt rt => AddTree.subNode (reduce lt) (reduce rt)
```

Listing 5.3.4: A natural map to reduce a double application of `AddTree` to a single application

Remark 5.3.1. In technical terms, this makes `AddTree` a *functor* from the category of types to itself. In fact, it also turns out to be a *monad* using 5.3.4 [FP20].

Actually recovering an `AddTree` representation of a concrete expression in Lean requires meta-programming; the snippet 5.3.5 contains a portion of the code together with an example at the end.

```
partial def treeM (e : Expr) : MetaM Expr :=
    hOp ``HAdd.hAdd e >>= (λ (a, b) => return ← mkAppM ``AddTree.node #[← treeM
    a, ← treeM b])    <|>
    hOp ``HSub.hSub e >>= (λ (a, b) => return ← mkAppM ``AddTree.subNode #[←
    treeM a, ← treeM b]) <|>
    invOp ``Neg.neg e >>= (λ a => return ← mkAppM ``AddTree.negLeaf #[← treeM
    a])                  <|>
    mkAppM ``AddTree.leaf #[e]

elab "treeElab" s:term : term => do
  let e ← Term.elabTerm s none
  treeM e
```

```
/- An example -/
eval treeElab -((2 + -3) - 1)
-- AddTree.negLeaf (AddTree.subNode (AddTree.node (AddTree.leaf 2)
    (AddTree.negLeaf (AddTree.leaf 3))) (AddTree.leaf 1))
```

Listing 5.3.5: A portion of the code used to process an expression in Abelian groups to produce an `AddTree` representing its syntax

For reasons of simplicity, we tackle the problem of *normalisation* of expressions in Abelian groups (i.e., converting a given expression such as $a+b-c-a+b$ into a *normal form expression* such as $0 \cdot a + 2 \cdot b + (-1) \cdot c$ in which every variable appears exactly once), which rests on the same ideas and is similar in spirit to the problem of solving equations in Abelian groups.

To convert an expression to its normal form, we must map it through the corresponding expression in a free Abelian group. We achieve this by defining the intermediate notion of an `IndexAddTree`, which just stands for `AddTree Nat` and encodes the positions of the individual variables in the expression; the corresponding `AddTree` in the free Abelian group can be produced from the `IndexAddTree` by substitution.

Once we have an expression in a free Abelian group $\mathbb{Z}^n$, we normalise it to its canonical form of an ordered tuple of integers by computational reduction. We finally round-trip back to the original Abelian group by mapping along the `inducedFreeMap` 5.3.6 from the Abelian group. The final step of showing that the original expression has the specified normal form requires some case-specific work that we have been unsuccessful in automating.

```
/-- The unique map ‘ℤ^n → A‘ taking the basis elements to the given list of
    values ‘l‘. -/
def inducedFreeMap {A : Type _} [AddCommGroup A] {n : ℕ} (l : List A) (h :
    l.length = n) : ℤ^n → A := ...
```

Listing 5.3.6: The unique map from a free Abelian group to a given list of elements in an Abelian group

5.4 Conclusion

The reduction of the general problem of proving an equation in all Abelian groups to the special case of the basis of the free Abelian group was certainly not specific to just Abelian groups, and it is likely that this kind of a reduction admits a more general formulation in the language of Category theory.

The correct analogue of a free Abelian group seems to be a *free object* in a category – the free objects in the category of groups are the free groups, the free objects in the category of rings are the polynomial rings, and so on.

Understanding this phenomenon in greater generality will likely reveal simpler ways to implement it in `Lean4`.

Bibliography

Asperti, Andrea and Jeremy Avigad. "Zen and the art of formalisation". In: *Mathematical Structures in Computer Science* 21.4 (2011), pp. 679–682.

Bentkamp, Alexander, Ramon Fernández Mir, and Jeremy Avigad. "Verified reductions for optimization". In: *arXiv preprint arXiv:2301.09347* (2023).

Bertot, Yves. "A short presentation of Coq". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, pp. 12–16.

Blanchette, Jasmin Christian, Lukas Bulwahn, and Tobias Nipkow. "Automatic proof and disproof in Isabelle/HOL". In: *Frontiers of Combining Systems: 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings 8*. Springer. 2011, pp. 12–27.

Bloom, Thomas F. "On a density conjecture about unit fractions". In: *arXiv preprint arXiv:2112.03726* (2021).

Carter, William. "New examples of torsion-free non-unique product groups". In: *Journal of Group Theory* 17.3 (2014), pp. 445–464.

Clarke, Edmund M, Manpreet Khaira, and Xudong Zhao. "Word level model checking—avoiding the Pentium FDIV error". In: *Proceedings of the 33rd annual Design Automation Conference*. 1996, pp. 645–648.

Community, The mathlib. "The Lean Mathematical Library". 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. 367–381. ISBN: 9781450370974. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.

Community, The mathlib. "The Lean Mathematical Library". 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. 367–381. ISBN:

9781450370974. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.

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

Dahmen, Sander R, Johannes Hölzl, and Robert Y Lewis. "Formalizing the solution to the cap set problem". In: *arXiv preprint arXiv:1907.01449* (2019).

De Bruijn, Nicolaas Govert. "A survey of the project AUTOMATH". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 133. Elsevier, 1994, pp. 141–161.

Diestel, Reinhard and Daniela Kühn. "Graph-theoretical versus topological ends of graphs". In: *Journal of Combinatorial Theory, Series B* 87.1 (2003), pp. 197–206.

Escardo, Martin. "Exhaustible sets in higher-type computation". In: *Logical methods in computer science* 4 (2008).

Fritz, Tobias and Paolo Perrone. "Monads, partial evaluations, and rewriting". In: *Electronic Notes in Theoretical Computer Science* 352 (2020), pp. 129–148.

Gardam, Giles. "A counterexample to the unit conjecture for group rings". In: *Annals of Mathematics* 194.3 (2021), pp. 967–979.

Geuvers, Herman. "Proof assistants: History, ideas and future". In: *Sadhana* 34 (2009), pp. 3–25.

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

Gonthier, Georges et al. "A machine-checked proof of the odd order theorem". In: *International conference on interactive theorem proving*. Springer. 2013, pp. 163–179.

Grushko, Igor Aleksandrovich. "On the bases of a free product of groups". In: *Matematicheskii Sbornik* 8.50 (1940), pp. 169–182.

Hales, Thomas et al. "A formal proof of the Kepler conjecture". In: *Forum of mathematics, Pi*. Vol. 5. Cambridge University Press. 2017.

Hales, Thomas C. *Mathematics in the age of the Turing machine*. 2014.

— "The Jordan curve theorem, formally and informally". In: *The American Mathematical Monthly* 114.10 (2007), pp. 882–894.

Halin, Rudolf. "Systeme Disjunkter Unendlicher Wege in Graphen". In: *Numerische Methoden bei Optimierungsaufgaben Band 3: Optimierung bei graphentheoretischen und ganzzahligen Problemen* (1977), pp. 55–67.

Harrison, John. "Formalizing an analytic proof of the prime number theorem". In: *Journal of Automated Reasoning* 43.3 (2009), pp. 243–261.

— "HOL light: An overview". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 60–66.

Higgins, PJ. "Grushko's theorem". In: *Journal of Algebra* 4.3 (1966), pp. 365–372.

Higman, G. "The Units of Group Rings. Londres". In: *Proc. London Math. Soc* 46 (1940).

Jech, Thomas. *Set theory: The third millennium edition, revised and expanded*. Springer, 2003.

Kaliszyk, Cezary and Freek Wiedijk. "Certified computer algebra on top of an interactive theorem prover". In: *Towards Mechanized Mathematical Assistants: 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007, Hagenberg, Austria, June 27-30, 2007. Proceedings*. Springer. 2007, pp. 94–105.

Kaplansky, Irving. ""Problems in the theory of rings" revisited". In: *The American Mathematical Monthly* 77.5 (1970), pp. 445–454.

La Harpe, Pierre de. *Topics in geometric group theory*. University of Chicago Press, 2000.

Limperg, Jannis and Asta Halkjær From. "Aesop: White-Box Best-First Proof Search for Lean". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2023, pp. 253–266.

Löh, Clara. *Geometric group theory*. Springer, 2017.

MacKenzie, Carlin, Jacques Fleuriot, and James Vaughan. "An Evaluation of the Archive of Formal Proofs". In: *arXiv preprint arXiv:2104.01052* (2021).

Mahboubi, A and E Tassi. *The Mathematical Components Libraries*. 2017.

Martin-Löf, Per and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

Massot, Patrick. "Why formalize mathematics?" In: (2021).

Massot, Patrick, Floris van Doorn, and Oliver Nash. "Formalising the h-principle and sphere eversion". In: *arXiv preprint arXiv:2210.07746* (2022).

Megill, Norman and David A Wheeler. *Metamath: a computer language for mathematical proofs*. Lulu. com, 2019.

Mikuła, Maciej et al. *Magnushammer: A Transformer-based Approach to Premise Selection*. 2023. arXiv: `2303.04488 [cs.LG]`.

Moura, Leonardo de and Sebastian Ullrich. "The lean 4 theorem prover and programming language". In: *International Conference on Automated Deduction*. Springer. 2021, pp. 625–635.

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

Murray, Alan G. "More counterexamples to the unit conjecture for group rings". In: *arXiv preprint arXiv:2106.02147* (2021).

Nagel, Ernest and James R Newman. "Gödel's proof". In: *NOTICES OF THE AMS* 51.3 (1988).

Naumowicz, Adam and Artur Korniłowicz. "A brief overview of Mizar". In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, pp. 67–72.

— "A brief overview of Mizar". In: *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer. 2009, pp. 67–72.

Nielsen, Jakob. "Om Regning med ikke-kommutative Faktorer og dens Anvendelse i Gruppeteorien". In: *Matematisk Tidsskrift. B* (1921), pp. 77–94.

Nipkow, Tobias, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

Nipkow, Tobias et al. *Functional Algorithms, Verified*. 2021.

Norell, U. *Dependently Typed Programming in Agda, Advanced Functional Programming, P. Koopman, R. Plasmeijer, and D. Swierstra (Eds.), Vol. 5832 of LNCS*. 2009.

Owre, Sam, John M Rushby, and Natarajan Shankar. "PVS: A prototype verification system". In: *Automated Deduction—CADE-11: 11th International Conference on Automated Deduction Saratoga Springs, NY, USA, June 15–18, 1992 Proceedings 11*. Springer. 1992, pp. 748–752.

Parillaud, Camille, Yoann Fonteneau, and Fabien Belmonte. "Interlocking formal verification at Alstom signalling". In: *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification: Third International Conference, RSSRail 2019, Lille, France, June 4–6, 2019, Proceedings 3*. Springer. 2019, pp. 215–225.

Passman, Donald S. "What is a group ring?" In: *The American Mathematical Monthly* 83.3 (1976), pp. 173–185.

Paulin-Mohring, Christine. *Introduction to the calculus of inductive constructions*. 2015.

Promislow, S David. "A simple example of a torsion-free, non unique product group". In: *Bulletin of the London Mathematical Society* 20.4 (1988), pp. 302–304.

Rijke, Egbert, Elisabeth Bonnevier, Jonathan Prieto-Cubides, et al. *Univalent mathematics in Agda*. `https://unimath.github.io/agda-unimath/`. URL: `https://github.com/UniMath/agda-unimath/`.

Robertson, Neil, Paul Seymour, and Robin Thomas. "Excluding infinite minors". In: *Discrete Mathematics* 95.1-3 (1991), pp. 303–319.

Scholze, Peter. “Liquid tensor experiment”. In: *Experimental Mathematics* 31.2 (2022), pp. 349–354.

Schreier, Otto. “Die untergruppen der freien gruppen”. In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*. Vol. 5. Springer. 1927, pp. 161–183.

Scott, Peter and Terry Wall. “Topological methods in group theory”. In: *Homological group theory (Proc. Sympos., Durham, 1977)*. Vol. 36. 1979, pp. 137–203.

Selsam, Daniel, Sebastian Ullrich, and Leonardo de Moura. “Tabled typeclass resolution”. In: *arXiv preprint arXiv:2001.04301* (2020).

Shankar, Natarajan. “Fixpoints and Search in PVS”. In: *Advanced Lectures on Software Engineering: LASER Summer School 2007/2008* (2010), pp. 140–161.

Sørensen, Morten Heine B and Pawe l Urzyczyn. “Curry-Howard Isomorphism”. In: *Univ. of Copenhagen, Univ. of Warsaw* (1998).

Stallings, John R. “A topological proof of Grushko’s theorem on free products”. In: *Mathematische Zeitschrift* 90.1 (1965), pp. 1–8.

Thurston, William P. “On proof and progress in mathematics”. In: *Bulletin of the American mathematical Society* 30.2 (1994), pp. 161–177.

Univalent Foundations Program, The. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.

Vaswani, Ashish et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

Whitehead, John HC. “Simple homotopy types”. In: *American Journal of Mathematics* 72.1 (1950), pp. 1–57.

Wiedijk, Freek. *The de Bruijn factor*. 2000.

Wu, Yuhuai et al. “Autoformalization with large language models”. In: *arXiv preprint arXiv:2205.12615* (2022).

Zach, Richard. “Hilbert’s program”. In: (2003).