Maintaining a Large Formalisation Project – 1

Part I: Mathematics & Infrastructure

Damiano Testa

University of Warwick

Tsinghua University, March 16-20, 2026

Section 1

Introduction

Who Am I

I am a mathematician, working on algebraic geometry and number theory.

Since 2020 I have also been interested in formalisation.

My experience with Lean comes from:

- individual formalisation projects
- participating in large collaborations
- being a Mathlib maintainer
- a project, sponsored by AI for Math, to formally verify computations performed by Macaulay2 — a computer algebra system focusing on commutative algebra and algebraic geometry computations

Experience

- **Formalisation**
  I contributed mostly to the algebraic and order areas of Mathlib.
- **Project-Scale Maintenance and Automation**
  I wrote some linters, as well as some tactics.
- **CI and Infrastructure**
  automated PR summaries, Zulip integration, deprecations.

Recent Projects

Currently, there are no public major projects that are mostly AI-driven.

Projects using a hybrid approach:

- the Equational Theory Project, led by Terry Tao
- the Sphere Packing Problem in Dimension 8 and 24, led by Maryna Viazovska and Sidharth Hariharan (arXiv)
- several small formalisations of Erdős Problems

If It Works for Humans...

What I will talk about applies to human-curated repositories.

I will focus on good practices with a view to what I expect would also be useful conventions for AI-generated content.

Principle

If something works for humans, maybe AI also benefits from it.

Large Formalisation Projects Involve a Mix of Expertise

Aspects to keep in mind:

- Mathematics — planning, development, corrections
- Automation of intermediate task assignments — parallelisability
- Vetting human contributions — code standards, maintainability
- Vetting AI contributions — same as above + guard against exploits
- Final merge decisions — PR quality
- Integration with current project — interoperability
- Re-usability of code-base — API, performance, maintainability
- Automatic detection — linters, naming, docs, guidelines
- CI ensuring main branch stability, caching, tests
- Mathlib bumps — fixes, improvements, refactors

Section 2

Mathematical Planning

Mathematics

Most larger formalisation projects start with mathematical planning.

Useful automation:

- track currently accessible targets
- past achievements
- partial progress
- task assignments
- milestones

Pre-Requisites

Some results may be required but out of scope.

For instance, the Feit–Thompson Theorem could be taken with a `sorry`.

Key Point

Statements must be formulated *very carefully*.

- Misformalisations can poison downstream development.
- AI agents may exploit small loopholes.

Section 3

Blueprints & Task Distribution

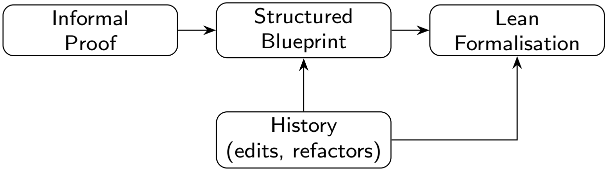
Maths Planning

Patrick Massot's blueprint keeps alignment between LaTeX source and formalisation.

As the formalisation progresses, the initial plan must be:

- revised
- adapted
- corrected
- expanded

Clear partial targets help parallelisation.



Distributing Tasks

- Each subproject/lemma/subsection can be assigned independently.
- Blueprint ensures all pieces fit together.
- Distributing large projects (human + AI) is one of the strengths of formalisation.

Section 4

Infrastructure & APIs

Mathematical Infrastructure: Generating API

Document and stabilise a small set of canonical interfaces, such as typeclasses, structures, notation.

This reduces churn and helps humans and AI.

Enforce good practice, such as

- avoid using `unfold` outside definition files
- break off long proofs
- generalise whenever possible

and so on.

Section 5

Quality & Misformalisations

Issues and Errors

Large projects uncover:

- typos
- omissions
- gaps
- errors
- restructuring needs

Fixes may need mathematical experts; small issues may have large ripple effects.

Bigger gaps, may require more serious restructuring.

Misformalisations

- Sometimes definitions differ from intended meaning.
- Some informal definitions do not suit dependent type theory.
- Need mathematically equivalent but formalisation-friendly formulations.
- Fixing an error may lead to proving a lemma that is unsuitable for later development.

$$\begin{tikzcd}
 & \boxed{\text{Misstated Definition}} \arrow[dl] \arrow[dr] & \\
\boxed{\text{Lemma 1}} \arrow[dr] & & \boxed{\text{Lemma 2}} \arrow[dl] \\
 & \boxed{\text{Broken Theorem}} &
\end{tikzcd}$$

Example: Correspondence Theorem

One of the most obvious examples of this is the ubiquitous presence of "there exists ..." statements in informal proofs.

Almost invariably, these purely existential statements hide an explicit construction.

For instance, the Correspondence Theorem for a group G and a normal subgroup N is stated as the *existence* of a bijection between the subgroups of G containing N and subgroups of the quotient G/N.

In practice, it is not the mere *existence* of such a bijection that is important, but rather that we can *explicitly write one down* and we can rely on specific properties of the given bijection.

This means that the "Correspondence Theorem" is really a `def`inition!

Section 6

Refactors & Reusability

Refactors

For human formalisations, refactors represent a thankless, but essential job.

It is unclear to me if refactors even mean the same for autoformalisation.

I expect that, at some scale, refactoring old code will be more efficient and beneficial than rewriting the whole project from the start.

Refactors should be:

- standalone PRs
- well-scoped
- accompanied by deprecation shims and migration notes

Refactors

Deprecations may not be needed, but there should be some mechanism for AI to not use the old API.

Changelogs and migration snippets help downstream files update smoothly.

Refactors may be required right after Mathlib bumps or milestone merges.

While this may not apply too much to AI-driven projects, automation for "common refactors" is lacking in the ecosystem but highly desired.

Re-Usability of Current Code-Base

For human-driven projects, reusability is crucial for:

- long-term maintainability
- enabling broad contributions
- avoid bloating of the code-base
- efficiency and performance

Definitions and lemmas should be general when appropriate, avoiding unnecessary specialisation to allow and encourage reuse.

Section 7

Maintaining Momentum

Mathlib Bumps

Mathlib evolves rapidly, but it is good to keep up to date!

Benefits:

- bug fixes
- new definitions and lemmas
- improved APIs
- more automation, tactics, diagnostics, performance, . . .

Downsides:

- break existing proofs
- rename constants
- restructuring of `import`s
- cause refactors

Side Projects

Useful for exploring:

- alternative definitions
- rethinking design decisions
- specialized tactics
- computational backends
- refactors
- training specialised agents

Fixing Gaps

Common reasons why a proof resists formalisation:

- the supplied proof is not detailed enough
- there are some missing hypotheses
- a previous lemma was not proved in enough generality
- there is a mistake in the argument

Hopefully this is a rare situation.

When it happens, “expert” intervention may be required.

From my perspective, this is a very exciting moment!

Thank You!

Questions?