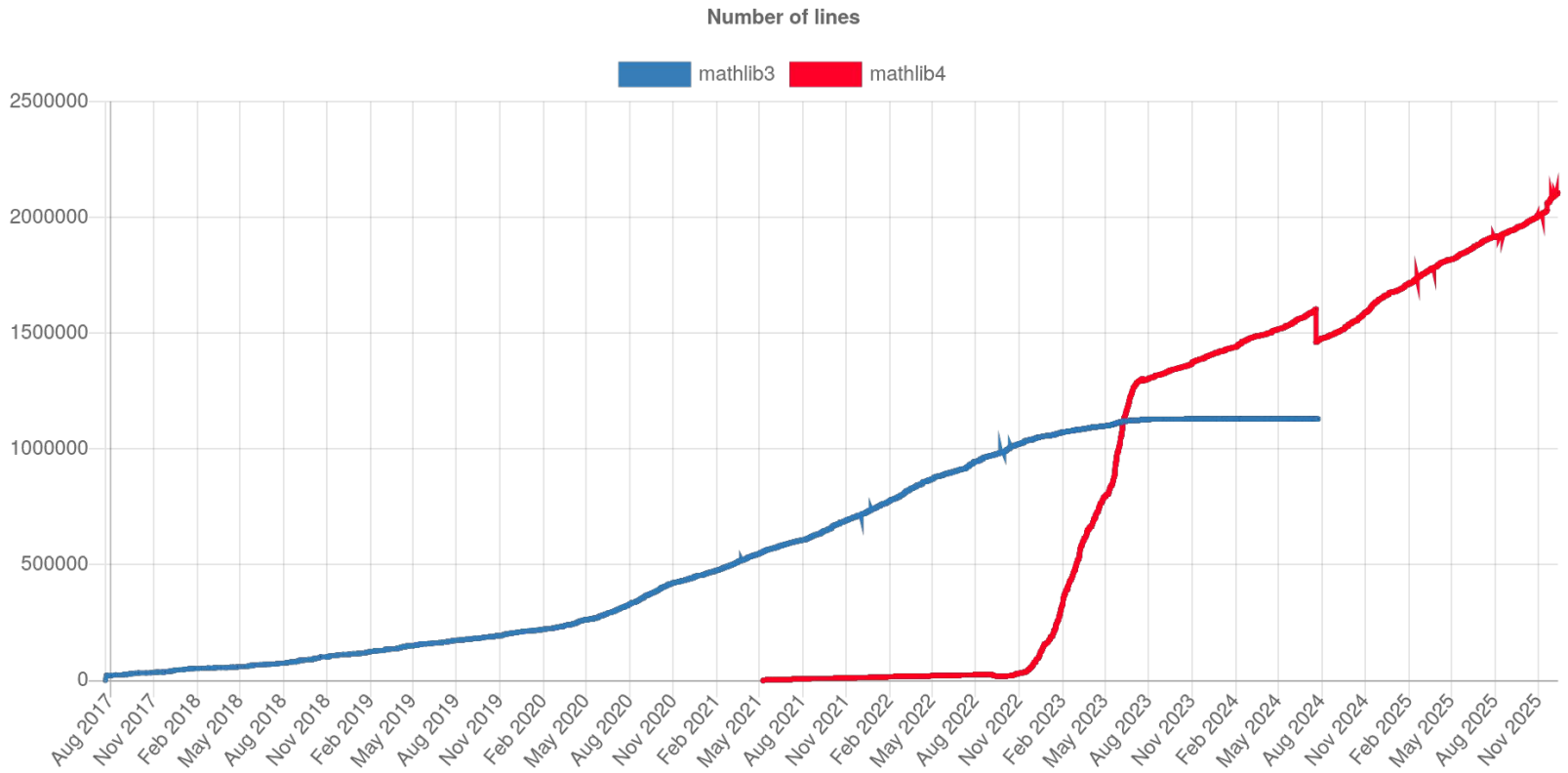
Optimizing Lean Libraries using the Module System

Sebastian Ullrich, Lean FRO
Lean@Google, Dec 15, 2025

Mathlib growth, never-ending…?



Many issues coming with that scale

- build times, locally and in CI, worst case and average case
  - 21min for full build on AMD 7950X3D with 32 hardware threads, ~27x speedup
  - ~30 master commits per day, plus many PR runs
  - dedicated CI machines provided by the Hoskinson Center (CMU)

Many issues coming with that scale

- build times, locally and in CI, worst case and average case
- output file size
  - 4.5GB on disk for full build, compressed to 270MB for network transfer
  - cloud cache provided by the Lean FRO, storing 7.3TB and serving ~10TB monthly

Many issues coming with that scale

- build times, locally and in CI, worst case and average case
- output file size
- human scalability
  - 

  - 442 contributors to date (mathlib4 only)
  - supported by a volunteer team of 26 maintainers
  - How do you robustly design a consistent library across 7000+ files?

Many issues coming with that scale

- build times, locally and in CI, worst case and average case
- output file size
- human scalability
  - 1,645 Open ✓ 24,537 Closed
  - 442 contributors to date (mathlib4 only)
  - supported by a volunteer team of 26 maintainers
  - How do you robustly design a consistent library across 7000+ files?

The central issue

Basic dependent typing is anti-modular!

```
def n : Nat := 0

theorem t : n = 0 := rfl
```

How far apart may `n` and `t` live?

Without the module system: *…however far apart they please*

Consequences of unrestricted reduction

All produced data is relevant, has to be imported by downstream files

- Including data transitively imported!
- No build short-circuiting on changes "irrelevant" to downstream files
- No way to exclude data from cache fetching
- No enforced API boundaries, no distinction between "interface" and "implementation"
- Too easy to do "accidental" unfolding
  - as in "you should have used that theorem instead"
  - as in not productive because it was unnecessary, potentially even backtracked

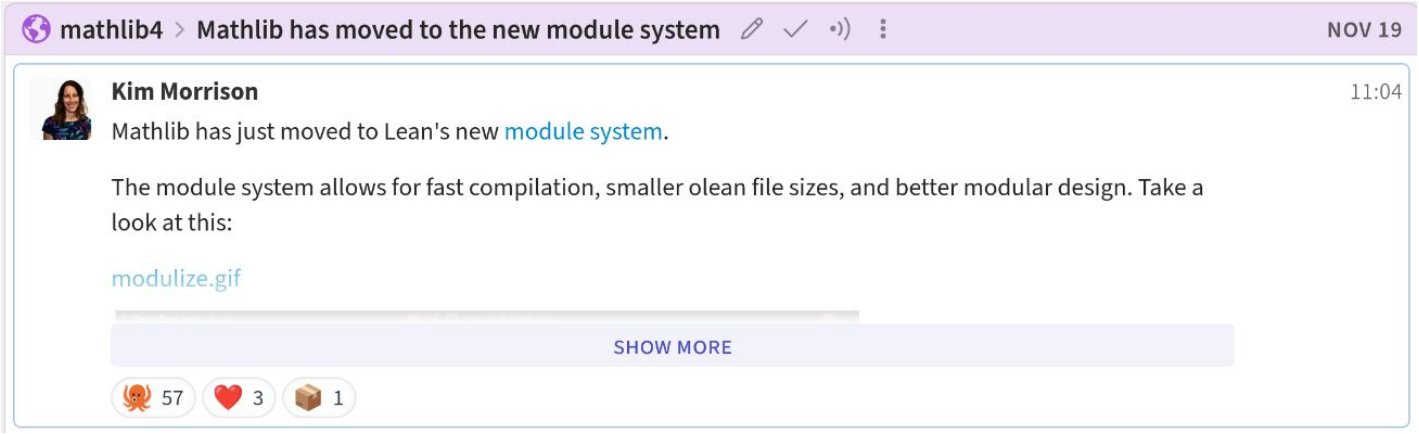
The Module System

A system of language restrictions to address the highlighted scalability issues

Experimental version since Lean 4.22, stabilized in 4.27.0-rc1 released **yesterday**

Opt-in via `module` header keyword, to be introduced top-down

Fully adopted by `Init`, `Std`, `Lean`, and `Mathlib` + dependencies



Module system basics: visibility

The *public scope* contains information to be exposed to other modules

Any other information is considered to live in the *private scope*

The public scope may not reference the private scope

```
public def n : Nat := 0

public theorem t : n = 0 := rfl
```

NB: default visibility has changed to private to encourage minimizing public decls

Module system basics: visibility

- Proofs (theorem bodies, nested `by`s) always live in the private scope
- `public def` bodies are private by default unless marked `@[expose]`
- `public abbrev` bodies are always public
- …

The Lean Language Reference

5.4. Modules and Visibility

Module system basics: extended imports

`public import A` imports the public scope of `A` into the public scope

`import A` imports the public scope in the private scope

- Thus `A` becomes irrelevant to any downstream users!

`import all A` imports both scopes into the private scope

- Primary use case: separating definitions and proofs for internal organization

Module system basics: phase separation

Data for *compile-time code execution* is necessarily transitive

Metaprograms in the module system are required to be tagged `meta`, cannot reference non-`meta` definitions and vice versa

`meta import` shifts imported defs into the `meta` phase

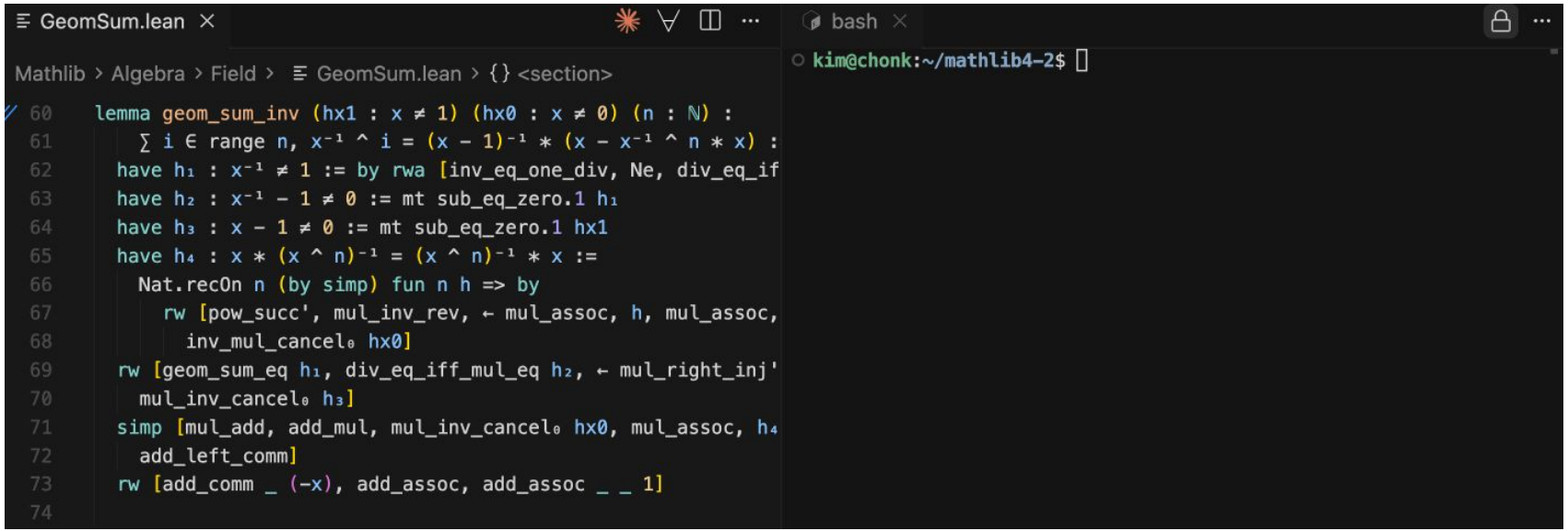
The Lean Language Reference

5.4.1. The Meta Phase

Module system goals revisited

Build times scalability

Changes limited to private scope => no need to recompile anything else



Module system goals revisited

Build times scalability

Changes limited to private scope => no need to recompile anything else

Public changes propagate through `public imports` but not beyond!

Module System Goals

Output size scalability

By default only public scope has to be (down)loaded [download **TBD** Q1'26]

=> load size decoupled from library source size

`Mathlib` breakdown:

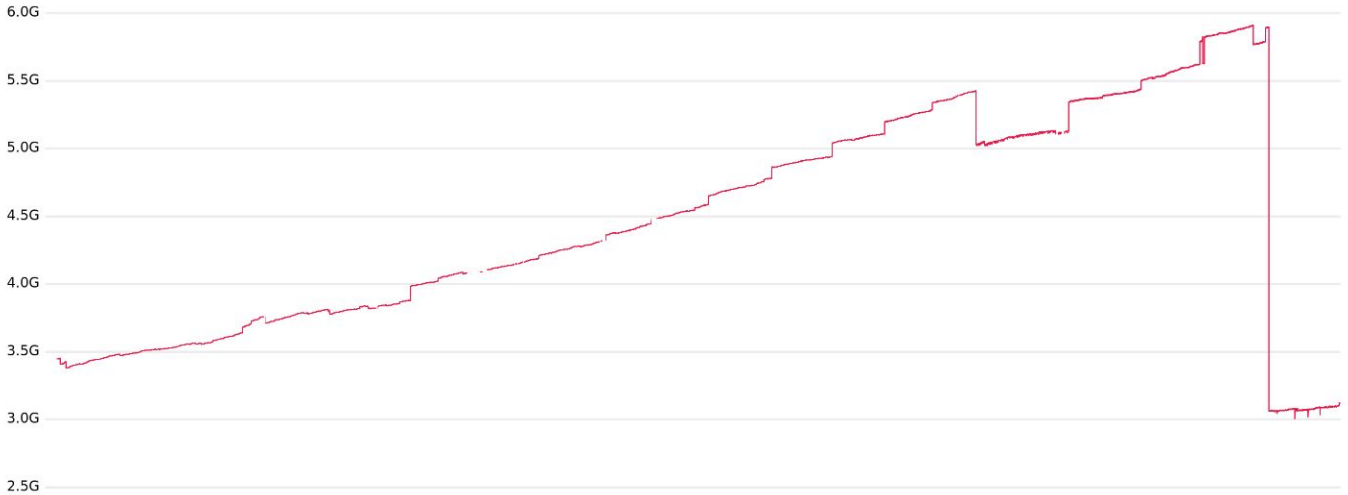
1.9GB public scope data
+ 0.2GB IR data for metaprogram execution
+ 0.1GB metadata for language server
+ 2.8GB private scope data

Module System Goals

Output size scalability

By default only public scope has to be (down)**loaded**

Pleasant surprise: RAM use of loading `Mathlib` fell by 48% from this



Module System Goals

Library design scalability

Changes in the private scope cannot influence other code

chore(CStarAlgebra): only import Liouville's theorem privately (mathlib4#32561)

Major changes (1)

- ✅ `build/module/Mathlib.Analysis.CStarAlgebra.ApproximateUnit//instructions`: **-14.9G** (**-7.9%**)

Module System Goals

Unfolding scalability

Enforced through private scope barrier and enticed by new `def` defaults

perf(Analysis.Complex.Exponential): make `Complex.exp` an `irreducible_def` (mathlib4#32599)

Major changes (3)

- ✅ `build/module/Mathlib.Analysis.Complex.UpperHalfPlane.Metric//instructions`: **-56.3G (-64.9%)**
- ✅ `build/module/Mathlib.Analysis.SpecialFunctions.Trigonometric.Basic//instructions`: **-13.2G (-14.5%)**
- ✅ `build/module/`

Module System Goals

Native code size scalability

`meta` marker will make it easier to strip away compile-time-only code, the largest contributor to the issue.

Deeper module system/codegen integration will be my focus of Q1 2026.

Further Module System Avenues

Native execution of custom tactics

`precompileModules` suffers from two issues:

- not compatible with Mathlib cache (to be integrated into Lake)
- had no idea what could be a metaprogram

What if we restricted it to `meta def`s?

Benefit/cost could even be high enough to do this by default

How to Port 2M Lines to the Module System

`script/Modulize.lean` automates the header syntax changes

`public @[expose] section` to mostly preserve remaining file's semantics

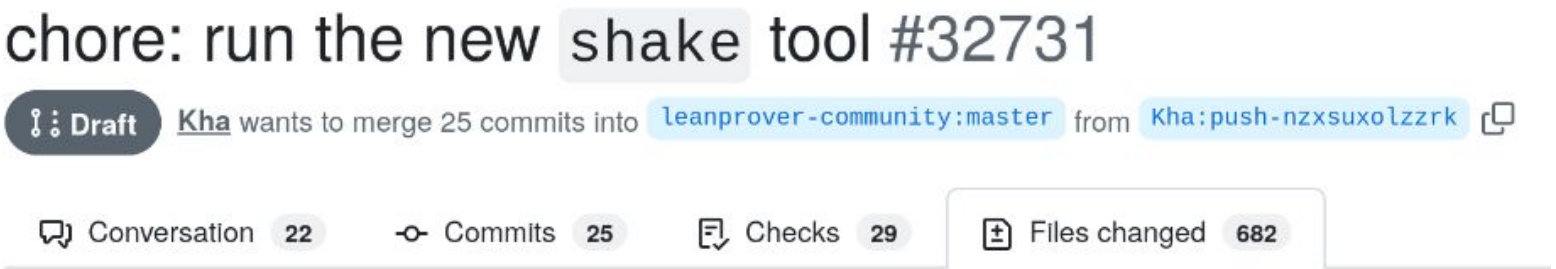
Metaprograms need `meta` annotations, sometimes reorganizing and adjusting

Prior uses of `private` unlikely to conform to new scope checks

- `backward.privateInPublic` keeps them in the public scope

Cleaning up with shake

New version of Mathlib's import minimizer: understands both module system imports and implicit metaprogramming dependencies



Conclusion

The module system introduces enforced information hiding to Lean

Benefits to API design, compilation speed & size, and disk & memory use

Now stable in the latest RC!