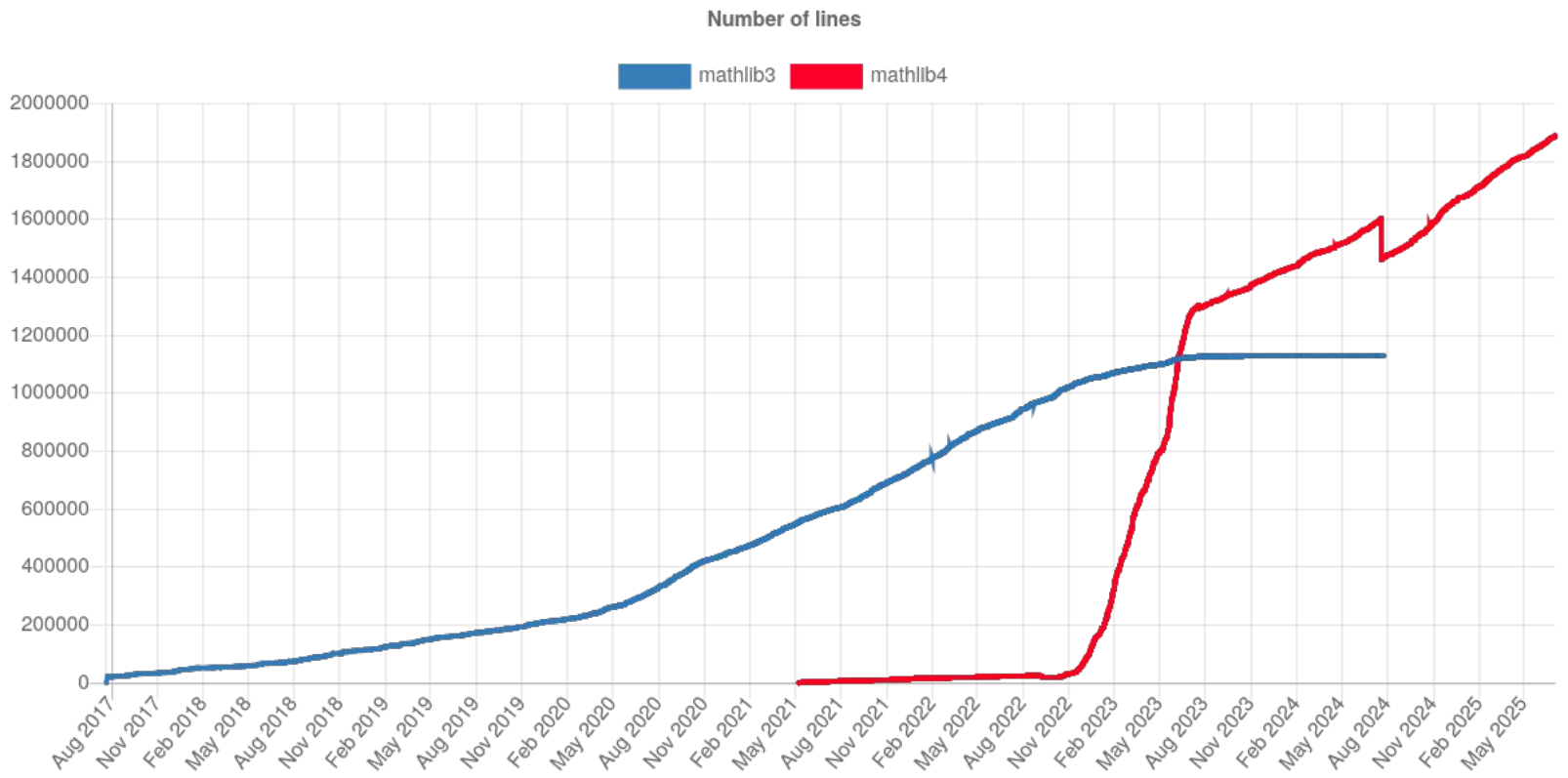
Library scalability in the Lean theorem prover

Sebastian Ullrich, Lean FRO
NUS, July 8, 2025

Ongoing Lean FRO projects

- much-improved error messages
- module system
- coinductive predicates
- monadic code reasoning
- new code generator
- Verso improvements
- Verilog frontend
- `grind` SMT-like automation
- Lake caching
- stdlib: ordering, strings, async I/O
- tactics for mathematicians, Mathlib tech debt
- …

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?

Status quo: *…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

Relevant related work

Controlling unfolding in type theory

Daniel Gratzer[1][0000−0003−1944−0789], Jonathan Sterling[1][0000−0002−0585−5564], Carlo Angiuli[2][0000−0002−9590−3303], Thierry Coquand[3][0000−0002−5429−5153], and Lars Birkedal[1][0000−0003−1320−0098]

[1] Aarhus University
[2] Carnegie Mellon University
[3] Chalmers University

Abstract. We present a novel mechanism for controlling the unfolding of definitions in dependent type theory. Traditionally, proof assistants let users specify whether each definition can or cannot be unfolded in the remainder of a development; unfolding definitions is often necessary in order to reason about them, but an excess of unfolding can result in brittle proofs and intractably large proof goals. In our system, definitions are by default not unfolded, but users can selectively unfold them in a local manner. We justify our mechanism by means of elaboration to a core type theory with *extension types*, a connective first introduced in the context of homotopy type theory. We prove a normalization theorem for our core calculus and have implemented our system in the `cooltt` proof assistant, providing both theoretical and practical evidence for it.

The Module System

A system of language restrictions to address the highlighted scalability issues

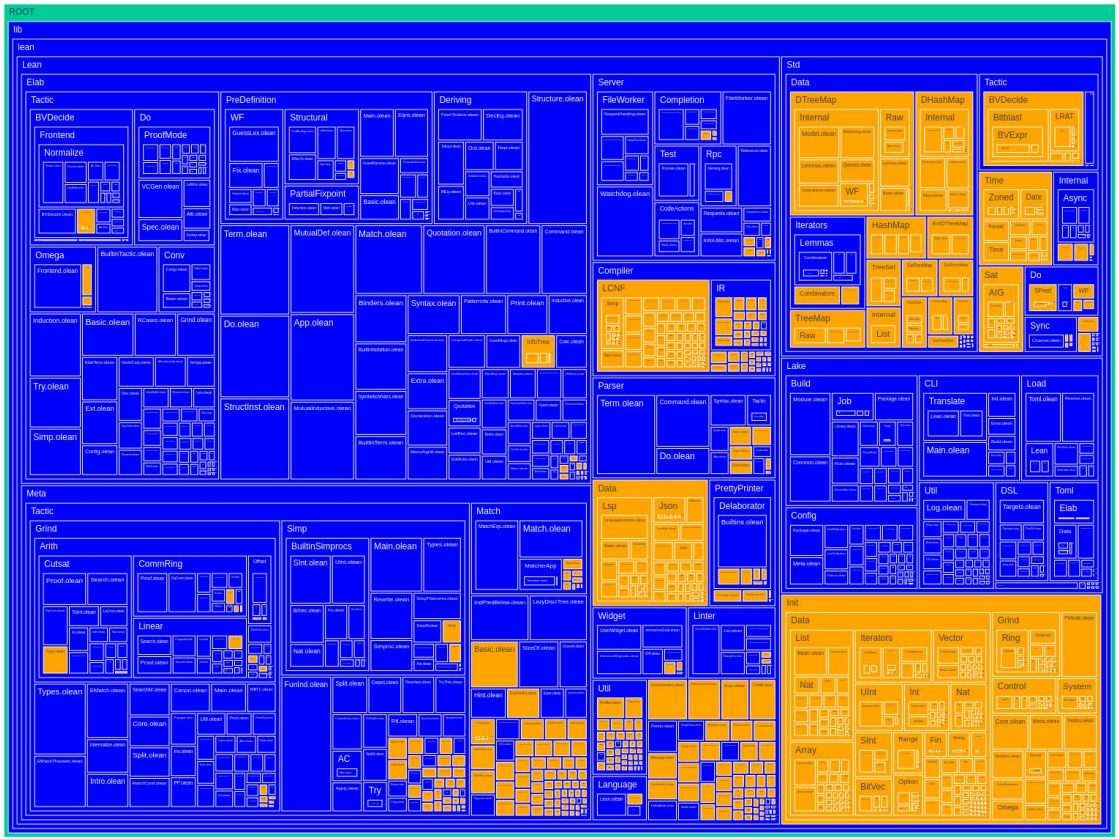
Experimental version available in Lean 4.22.0-rc

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

Current status:

- extending coverage from `Init` to `Std` and `Lean`
- bug fixes and UX improvements
- reworking internal metadata to utilize the new restrictions

The Module System



Module system basics: visibility

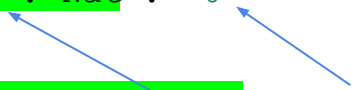
The *public scope* contains information to be exposed to other modules

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
```



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
- …

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

- But only within a library
- Allows for separating definitions and proofs internally

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 imported non-`meta` definitions

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

Module system goals revisited

Build times scalability

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

Transitive public changes shifted to private scope via non-`public` import => no need to recompile anything from here on

Module System Goals

Output size scalability

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

=> load size decoupled from library source size

`Init` breakdown:

71MB public scope data
+ 6.2MB metadata for language server
+ 156MB private scope data

Module System Goals

Library design scalability

Changes in the private scope cannot influence other code

- except through `import all`, which is limited to the same library

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

Module System Goals

Unfolding scalability

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

To be tested and measured in vivo

Learnings from porting, so far

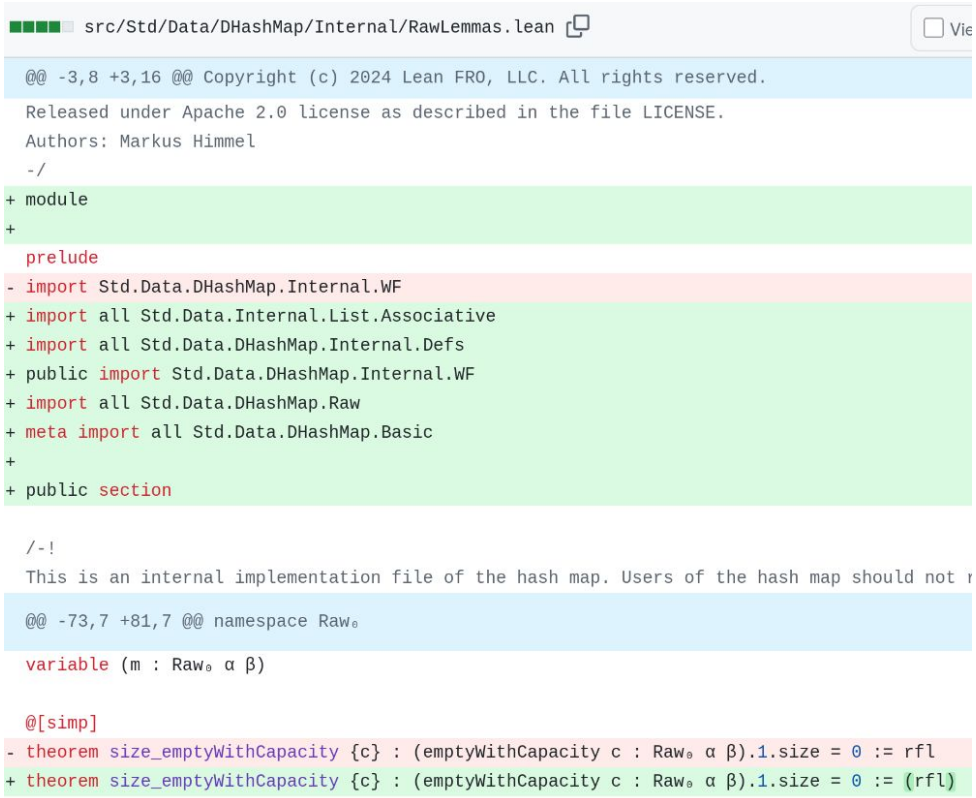
Purely syntactic changes can be automated away

Start with `public @[expose] section` to avoid most of that churn

Largest hurdles need human decision making:

- Something stops reducing, `@[expose]` the original or use `import all`?
- Special case `@[simp] … := rfl` theorems: keep definitional equality or export propositionally only?
- A few `meta`s have to be added
- Most previous uses of `private` now ill-scoped, need restructuring

Final Appearance



Demo

Conclusion

The module system introduces enforced information hiding to Lean

Benefits to API design, compilation speed, and disk use

Now available for early experimentation

Expect `Std`+`Lean` ported and more improvements in the next release!