Scaling Lean to the next Millions of Lines of Proofs

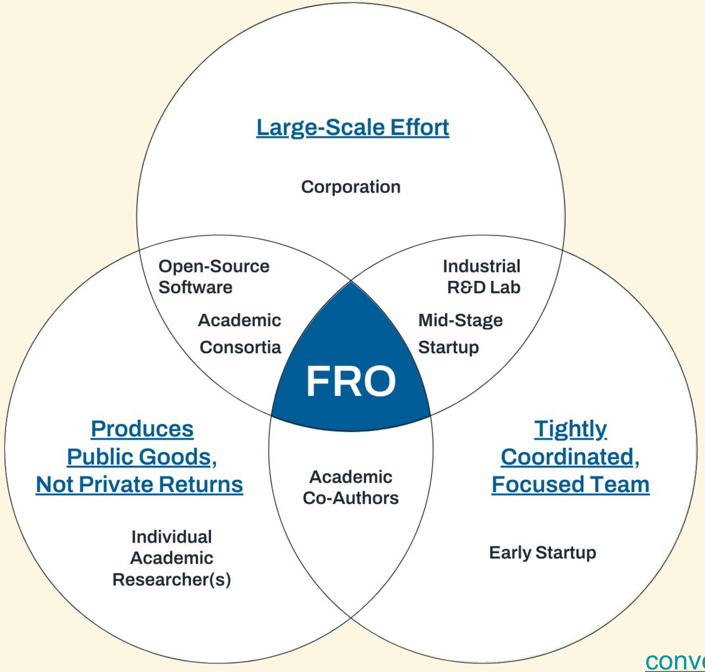
Sebastian Ullrich (Lean FRO)

LEAN
THEOREM PROVER

leanprover.github.io/publications/

Focused Research Organization (FRO)

A new type of nonprofit startup for science developed by Convergent Research



convergentresearch.org

The Lean FRO

Mission: address scalability, usability, and proof automation in Lean

~7 FTEs by end of year

Supported by Simons Foundation International, Alfred P. Sloan Foundation, and Richard Merkin

lean-fro.org

The Lean FRO

Mission: address scalability, usability, and proof automation in Lean

~7 FTEs by end of year

Supported by Simons Foundation International, Alfred P. Sloan Foundation, and Richard Merkin

lean-fro.org

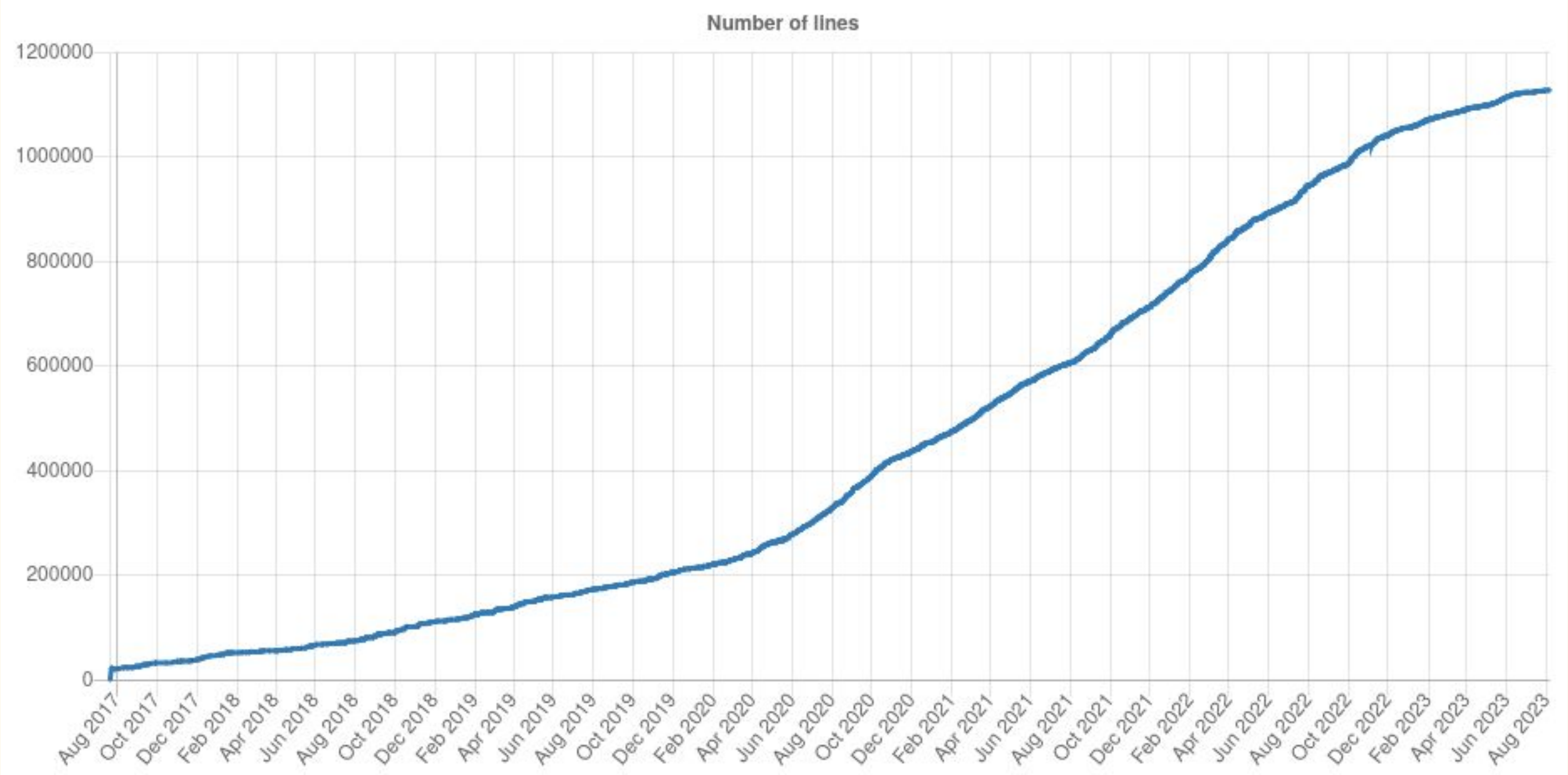
Questions of Scale

“Can mathlib scale to 100 times its present size, with a community 100 times its present size and commits going in at 100 times the present rate? [...] Will the proofs be maintained afterwards [...]?”

– Joseph Myers on Lean Zulip

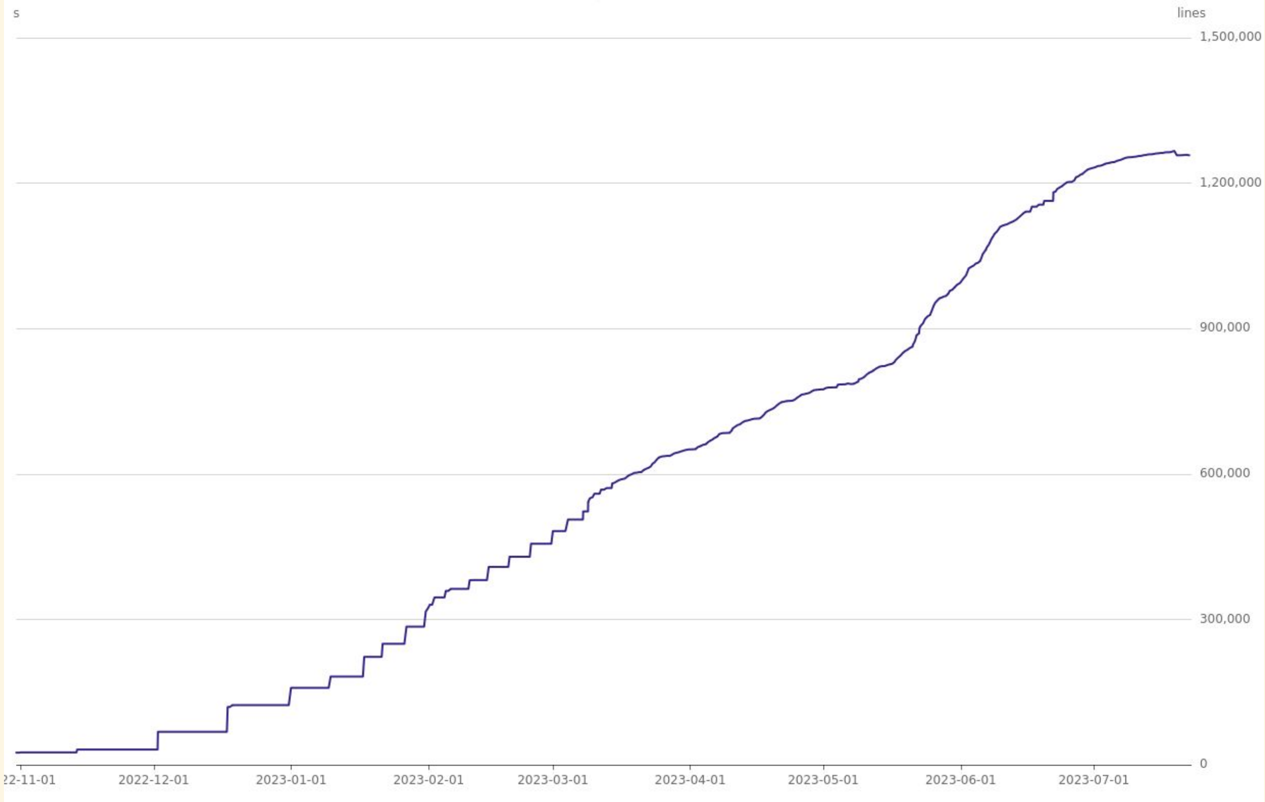
Part 1: Status Quo

Mathlib Growth

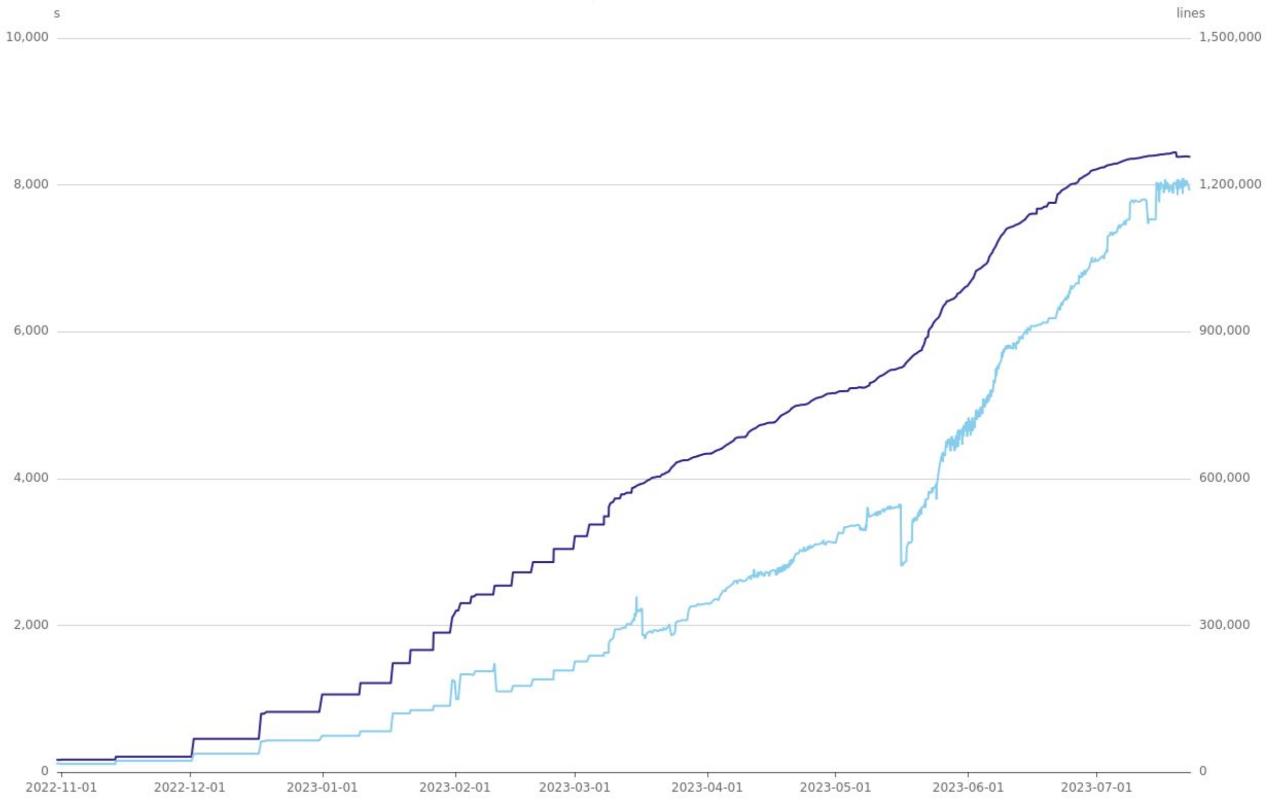


leanprover-community.github.io/mathlib_stats.html

The Mathlib Port

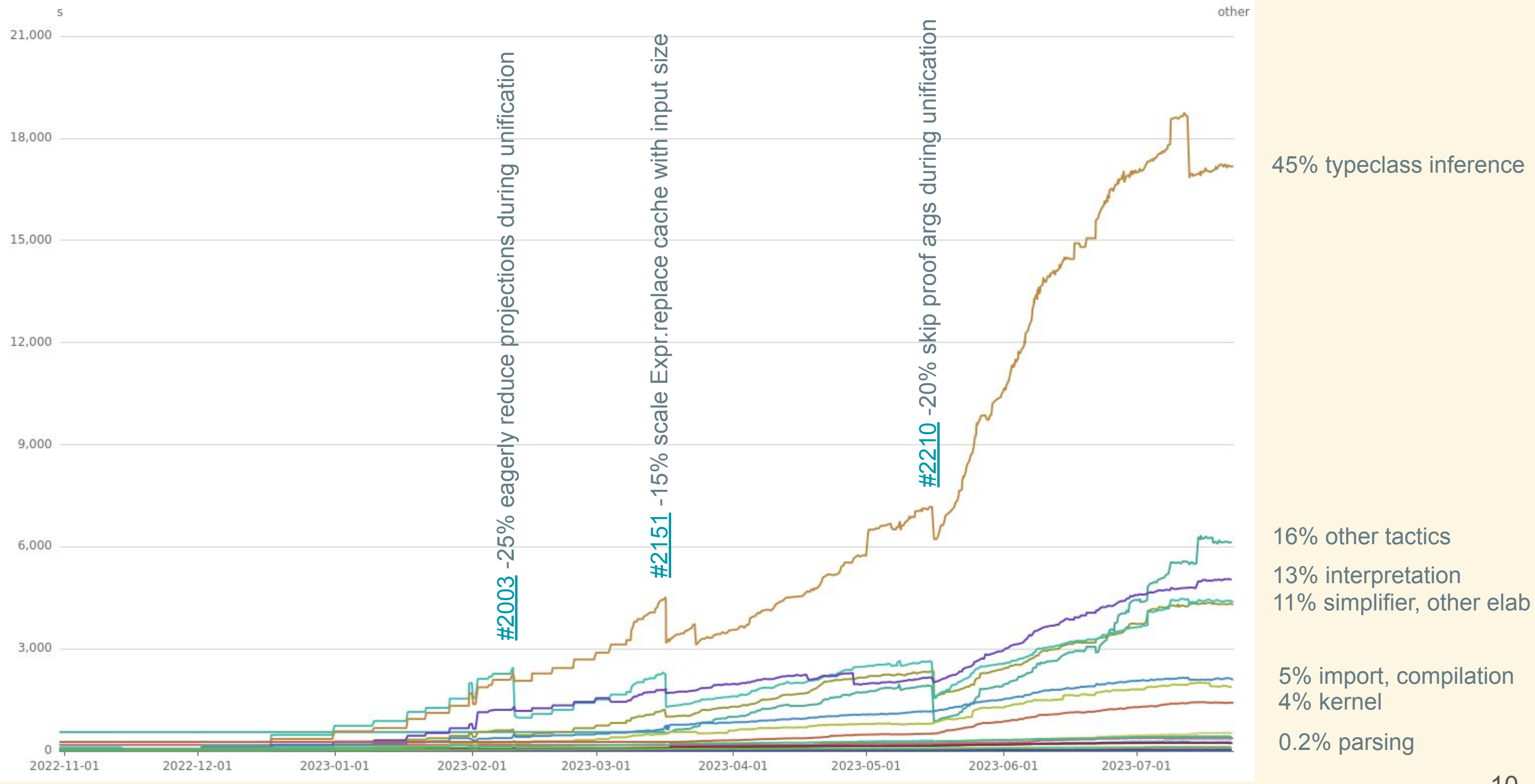


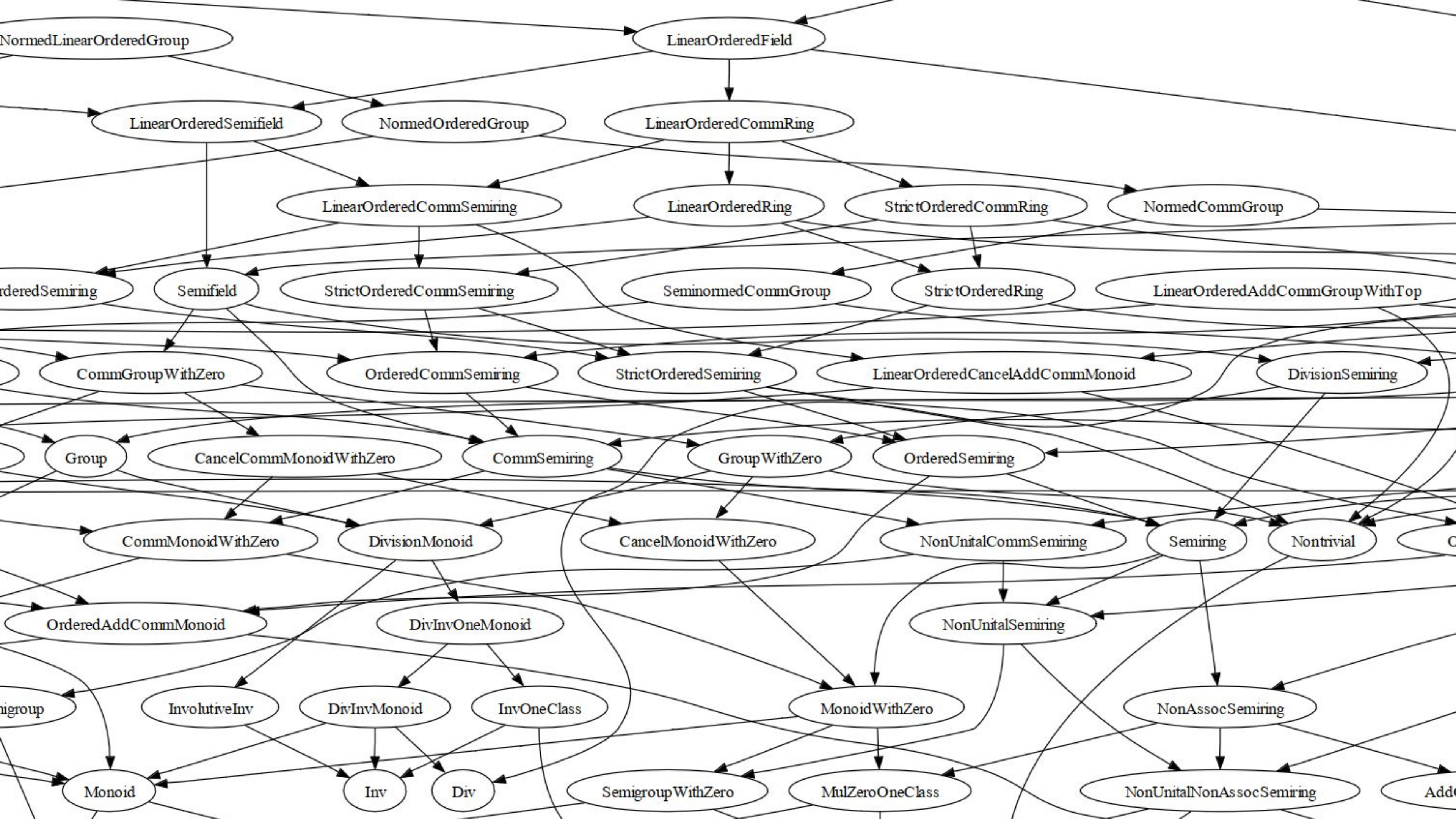
The Mathlib Port: Build Times



speed.lean-fro.org/mathlib4

The Mathlib Port: Breakdown into Categories





NormedLinearOrderedGroup
LinearOrderedField
LinearOrderedSemifield
NormedOrderedGroup
LinearOrderedCommRing
LinearOrderedCommSemiring
LinearOrderedRing
StrictOrderedCommRing
NormedCommGroup
Semifield
StrictOrderedCommSemiring
SeminormedCommGroup
StrictOrderedRing
LinearOrderedAddCommGroupWithTop
CommGroupWithZero
OrderedCommSemiring
StrictOrderedSemiring
LinearOrderedCancelAddCommMonoid
DivisionSemiring
Group
CancelCommMonoidWithZero
CommSemiring
GroupWithZero
OrderedSemiring
CommMonoidWithZero
DivisionMonoid
CancelMonoidWithZero
NonUnitalCommSemiring
Semiring
Nontrivial
OrderedAddCommMonoid
DivInvOneMonoid
NonUnitalSemiring
InvolutiveInv
DivInvMonoid
InvOneClass
MonoidWithZero
NonAssocSemiring
Monoid
Inv
Div
SemigroupWithZero
MulZeroOneClass
NonUnitalNonAssocSemiring

Performance: Before (Lean 3) and After (Lean 4)

On a Ryzen 9 (32 threads):

Total build time: 48 min ~> 21 min (-55%)

Single-core time: 23 hours ~> 5 hours (-77%)

Typeclass inference: 3 hours ~> 1 hour 46 min (-42%)

Performance: Importing Mathlib

disk: 436 MB ~> 3.1 GB (+711%)

time: 10.6 s ~> 1.5 s (-86%)

allocations: 4.6 GB ~> 243 MB (-95%)

due to zero-cost deserialization via memory mapping

Part 2: Challenges

Automation is Hard

Current and future bottleneck is clearly automation, >70% of current build time

Lean 4 *discrimination tree* essential for avoiding unification during search

Tabled resolution avoids redundant goals in typeclass inference

Ultimately an open-ended problem

What Do We Want to Measure?

Time for full rebuild is simple, but more relevant metrics in practice would be

- time of incremental build
- time to see the effect of a change

Current Lean 4 Build Model

File level: standard LCF-style pipeline: parse, process, and kernel-check declaration by declaration. *No parallelism.*

Package level: build dependency graph from (transitive) import declarations, process in parallel. *No short-circuiting.*

Part 3: Plans and Dreams

Where to Even Begin

More parallelism gets us linear speedup, increasing each year. That’s nice.

Build short-circuiting can reduce a global rebuild to a limited local one.

That’s great.

Build Short-Circuiting

Easy: recompile dependents only when *really* affected by a change

- C, C++, ML: write public interface of implementation file manually

  Coq: A Case for Lightweight Interfaces in Coq [Swasey et al. 2022] proposal

- GHC: automatically derive interface from file contents
- Rust: track fine-grained dependencies of disk-memoized queries

Build Short-Circuiting

Easy: recompile dependents only when *really* affected by a change

- C, C++, ML: write public interface of implementation file manually

  Coq: A Case for Lightweight Interfaces in Coq [Swasey et al. 2022] proposal

- GHC: automatically derive interface from file contents
- Rust: track fine-grained dependencies of disk-memoized queries

Towards a Lean Interface

- Signatures of public declarations

```
private def merge [Ord α] (xs ys : Array α) : Array α := …

def sort [Ord α] (xs : Array α) : Array α := match xs with …

theorem sort_sorted : Sorted (sort xs) := by …
```

- No proofs. Irrelevant anyway!

- No definition bodies or equations *by default*
  - A file-level Controlling unfolding in type theory [Gratzer et al. 2022]
  - *abbreviations*, definitions to be inlined always included

Cutting the Import Knot

Private imports are not part of the signature

```
import Mathlib.Algebra.Ring
private import Mathlib.Data.Real.CauSeqCompletion

def Real : Type := CauSeq.Completion.Cauchy (abs : ℚ → ℚ)
instance : Ring Real := …
```

Demotes public changes to private changes from this point on!

Metaprogramming Woes

Metaprogramming is anti-modular: promotes private changes to public

```
import Init.Data.Array.Sort for meta

macro "sorted" nums:num* : term =>
  let nums := nums.sort
  …
```

`meta` *phase* isolates code needed for build-time execution

Metaprogramming Woes

Metaprogramming is anti-modular: promotes private changes to public

```
import Init.Data.Array.Sort for meta

macro "sorted" nums:num* : term =>
  let nums := nums.sort
  …
```

`meta` *phase* isolates code needed for build-time execution

But what about a quick `#eval #[2, 1].sort`?

Interactive use might want to be more lenient

Transitioning

How do we move 1M+ lines to this model? Incrementally!

- Keep `import` semantics as is, disregarding annotations upstream
- Introduce `import signature` command for restricted behavior, adapt files top-down

Usability

Specifying fine-grained imports is clearly more work!

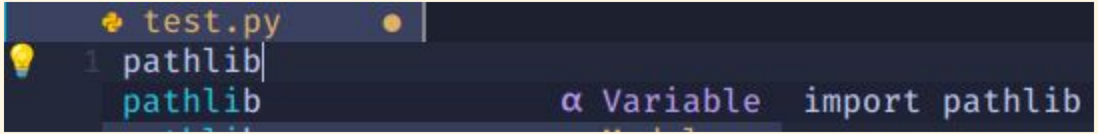
For new files *and transitioning*, tooling to reduce coarse imports would be great

Usability

Specifying fine-grained imports is clearly more work!

For new files *and transitioning*, tooling to reduce coarse imports would be great

For modifying existing files, language server should offer options outside current imports as well

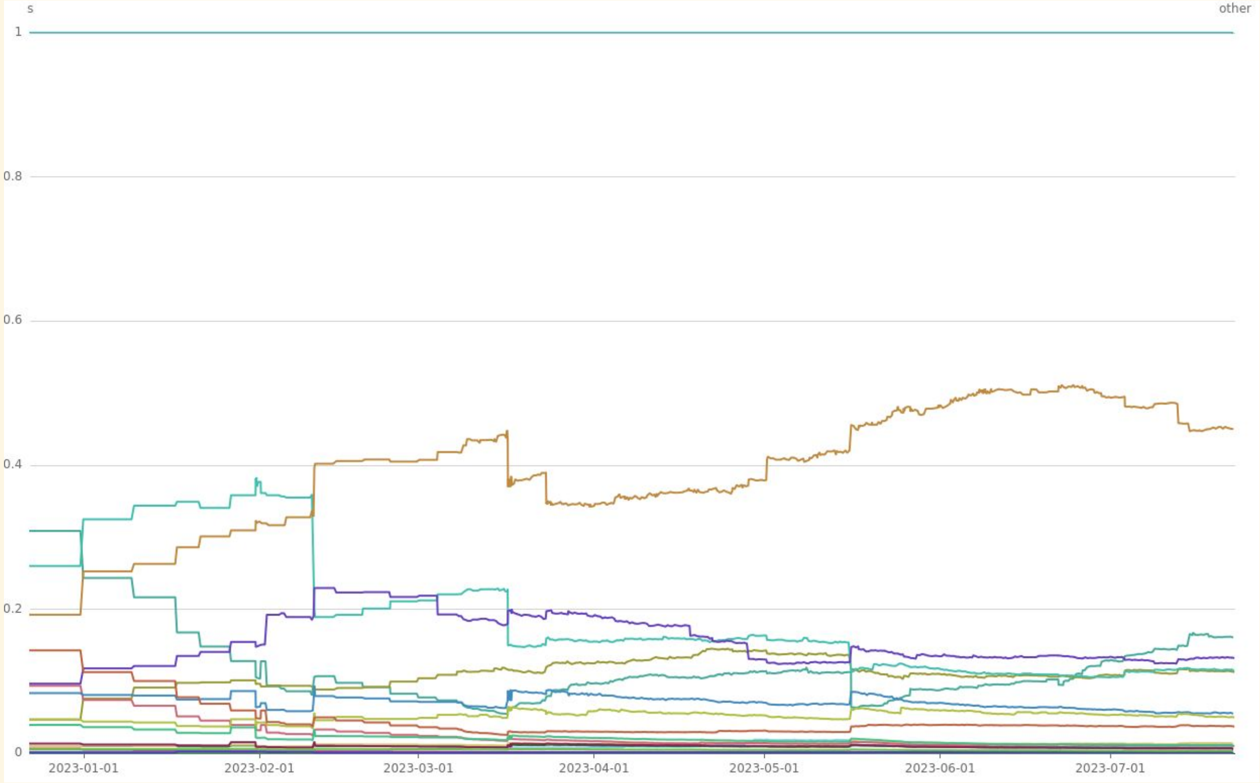


Summary

Lean 4 brings significant improvements to scalability over its predecessors

Modularity and abstraction will be key for uncoupling resource use and code growth

Categories normalized by task-clock



More Related Work

- Isabelle can postpone/parallelize proof checking across files
- so can Coq quick-compile
- iCoq [Celik et al. 2017] tracks dependencies for *regression proof selection*