Scaling Lean to the next Millions of Lines of Proofs

Sebastian Ullrich (Lean FRO)
Focused Research Organization (FRO)

A new type of nonprofit startup for science developed by Convergent Research
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](https://lean.zulip.org)
Part 1: Status Quo
Mathlib Growth

[Graph showing the growth of Mathlib over time, with a significant increase in the number of lines from August 2017 to August 2023.]
The Mathlib Port
The Mathlib Port: Build Times

[speed.lean-fro.org/mathlib4]
The Mathlib Port: Breakdown into Categories

45% typeclass inference
16% other tactics
13% interpretation
11% simplifier, other elab
5% import, compilation
4% kernel
0.2% parsing
Performance: Before (Lean 3) and After (Lean 4)

On a Ryzen 9 (32 threads):

Total build time: 48 min $\rightarrow$ 21 min (-55%)

Single-core time: 23 hours $\rightarrow$ 5 hours (-77%)

Typeclass inference: 3 hours $\rightarrow$ 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 \textit{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: \textit{automatically derive interface from file contents}

- Rust: track fine-grained dependencies of disk-memoized queries
Towards a Lean Interface

● Signatures of public declarations

```plaintext
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](https://doi.org/10.1145/3544662.3544671) [Gratzer et al. 2022]
  ○ *abbreviations*, definitions to be inlined always included
Cutting the Import Knot

*Private* imports are not part of the signature

```lean
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

```haskell
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