Profiling Tools in Lean

Sebastian Ullrich, Lean FRO
LMU, 2024/06/06

The Basics: `lean --profile`

```
~/lean/mathlib4  master  lake env lean Mathlib/Algebra/Category/ModuleCat/Presheaf.lean --profile
import took 547ms
simp took 142ms
tactic execution of Lean.Elab.Tactic.Ext.applyExtTheorem took 232ms
simp took 133ms
simp took 298ms
simp took 227ms
simp took 291ms
elaboration took 110ms
simp took 148ms
simp took 574ms
elaboration took 169ms
simp took 174ms
simp took 589ms
compilation of PresheafOfModules.unitHomEquiv took 171ms
elaboration took 164ms
cumulative profiling times:
        aesop 409ms
        attribute application 21.3ms
        compilation 407ms
        dsimp 35.6ms
        elaboration 1.35s
        import 547ms
        initialization 18.9ms
        interpretation 314ms
        linting 57ms
        parsing 17.6ms
        simp 4.65s
        tactic execution 688ms
        type checking 869ms
        typeclass inference 2.78s
```

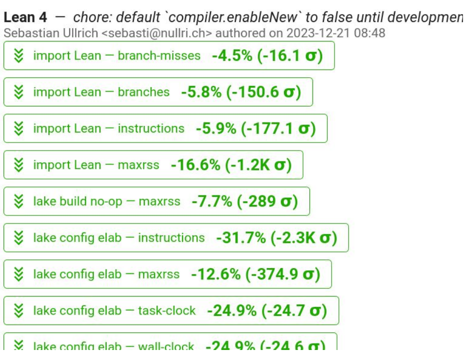
Coarse, exclusive (“self-time”) timing of selected components

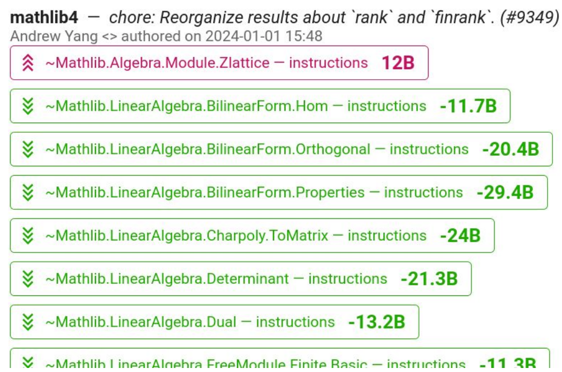
issues:

- no location information
- no hierarchical structure, which parts of simp are called by aesop?
    - also prevents us from refining categories, “interpretation” as a separate category not counted towards callers is already suspicious

The Basics, in the Cloud

Continuous benchmarking of each lean4 & mathlib4 commit

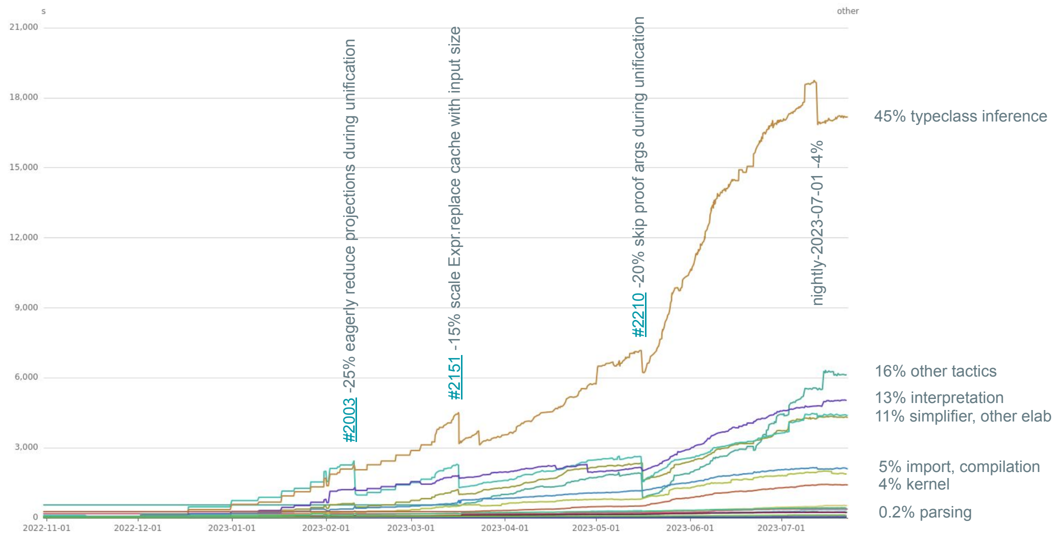




speed.lean-lang.org

speed.lean-lang.org provides these profiles for every commit, perfect for graphing and short- or long-time comparisons

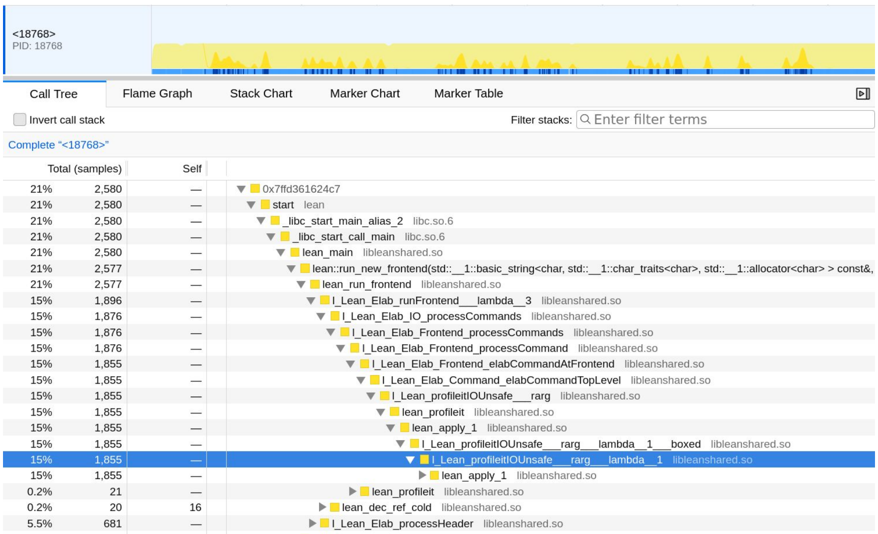
The Basics, in the Cloud



See also *Are We Fast Yet*, Lean Together 2024

Was very helpful to track growth of mathlib4 during the port as well as speed-ups from Lean changes; see my LT talk for details and future plans for making Lean faster

Traditional profiling: samply



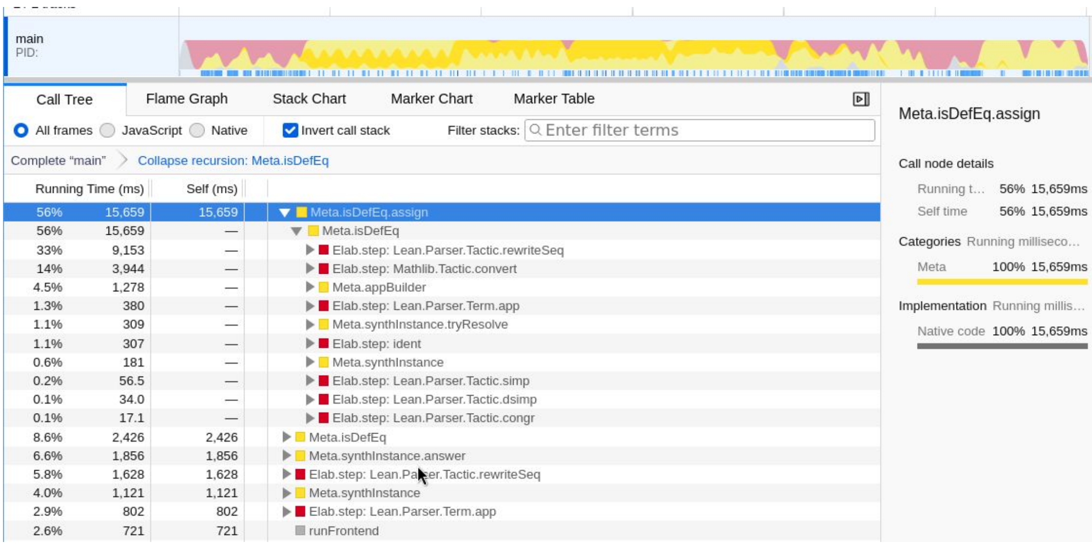
How can we get more structured information? Traditional profiling tools work surprisingly well with Lean in general because of its light runtime, but on a compiler like Lean itself they are far too low-level; they don't tell us about the structure of the input and are mainly useful for optimizing specific components of Lean itself. Another issue is that deep recursion in programs like Lean is not well supported and leads to stack traces being cut off (see "start" encompassing a mere 21% of the full run time).

On a Higher Level: `trace.profiler`

```
[Elab.command] [1.180741] instance : AddCommGroup (P ⟶ Q) [...]
▼
  [step] [0.117268] intros; ext1; simp only [add_app, add_assoc]
▼
    [] [0.117205] intros; ext1; simp only [add_app, add_assoc] ▼
      [] [0.115459] simp only [add_app, add_assoc] ▼
        [Meta.isDefEq] [0.060067] ✖ ?a + ?b + ?c =?= a✝ + b✝ +
c✝ ▶
        [Meta.isDefEq] [0.017167] ✅ ?a + ?b + ?c =?=
PresheafOfModules.Hom.app a✝ X✝ + PresheafOfModules.Hom.app b✝
X✝ + PresheafOfModules.Hom.app c✝ X✝ ▶
  [step] [0.105159] intros; ext1; simp only [add_app, zero_app,
zero_add] ▶
  [step] [0.082493] intros; ext1; simp only [add_app, zero_app,
add_zero] ▶
  [step] [0.094103] intros; ext1; simp only [add_app, sub_app,
neg_app, sub_eq_add_neg] ▶
  [step] [0.010252] intros; rfl ▶
  [step] [0.135185] intros; ext1; simp only [add_app, neg_app,
add_left_neg, zero_app] ▶
  [step] [0.243628] intros; ext1; simp only [add_app]; apply
add_comm ▶
  [Kernel] [0.019132] typechecking declaration
```

A very simple change turned out to provide much better data: the `trace.profiler` option takes the existing trace tree system used for debugging Lean and automatically activates and annotates any nodes above `trace.profiler.threshold` (10ms by default). This enabled users to find out for themselves where time in their proofs is actually being spent. However, a textual tree is still hard to scan for the critical path etc.

On a Higher Visual Level: `trace.profiler.output`



In Lean 4.8.0 we combined the strengths of traditional profiling and the trace profiler via an export option into Firefox Profiler, providing all its visualization and manipulation options

Mathlib-wide trace

Running Time (ms)		Self (ms)	
100%	22,115,628	788,076	▼ runFrontend
44%	9,678,013	950,403	▶ Elab.command: Lean.Parser.Command.theorem
18%	4,047,215	380,061	▶ Elab.command: Lean.Parser.Command.definition
13%	2,933,633	2,933,633	Import
7.1%	1,580,630	250,565	▶ Elab.command: Lean.Parser.Command.instance
6.2%	1,366,133	823	▶ Elab.command: lemma
2.2%	493,494	493,494	Elab.command: Mathlib.Prelude.Rename.align
2.1%	463,312	553	▶ Elab.command: Lean.Parser.Command.in
1.4%	318,487	79,952	▶ Elab.command: Lean.Parser.Command.variable
0.5%	115,274	27,123	▶ Elab.lint
0.3%	73,138	29,710	▶ Elab.command: Lean.Parser.Command.structure

share.firefox.dev/3PKliS1

Unlike with traditional profiling, the output is also coarse enough that we're not restricted to seconds-long programs but can profile all of Mathlib

Mathlib-wide trace, inverted

All frames · JavaScript · Native · Invert call stack (checked)

Complete “collided”

Running Time (ms)		Self (ms)	
19%	4,094,129	4,093,957	Meta.synthInstance
13%	2,933,633	2,933,633	Import
7.2%	1,590,020	1,589,091	Meta.isDefEq
6.9%	1,520,170	1,520,175	Elab.step: Lean.Parser.Term.app
6.1%	1,352,319	1,352,320	Elab.step: Lean.Parser.Tactic.simp
5.9%	1,310,211	1,310,211	Elab.command: Lean.Parser.Command.theorem
4.7%	1,044,699	1,044,699	aesop
3.6%	788,076	788,076	runFrontend
3.2%	706,316	706,316	compiler
2.5%	560,569	560,569	Kernel
2.3%	516,143	516,143	Elab.command: Mathlib.Prelude.Rename.align
2.2%	480,388	480,387	Elab.command: Lean.Parser.Command.definition
2.1%	464,716	464,709	Elab.step: Lean.Parser.Tactic.tacticSeq1Indented
1.7%	383,247	383,247	Elab.step: Lean.Parser.Tactic.rewriteSeq
1.5%	331,225	331,223	Meta.isDefEq.assign
1.4%	309,078	309,078	Elab.step: Lean.Parser.Term.binrel
1.3%	280,503	280,503	Elab.command: Lean.Parser.Command.instance
1.2%	276,312	276,312	Elab.header
1.2%	272,131	272,131	Elab.step: Lean.Parser.Tactic.simpa
1.1%	233,710	233,710	Meta.check
1.0%	228,673	228,673	Meta.Tactic.simp.discharge
0.9%	207,327	207,327	Elab.step: Mathlib.Tactic.convert

Inverting the “call stack” gives us a new view of the “hottest” components, i.e. with the highest self-time, but also with additional context when expanding nodes

Boiling it Down: `diagnostics`

```
[reduction] unfolded declarations (max: 101424, num: 17): ▼
  Nat.rec ↦ 101424
  Add.add ↦ 46937
  HAdd.hAdd ↦ 46937
  Nat.add ↦ 32191
  OfNat.ofNat ↦ 11060
  SetLike.coe ↦ 8158
  Set ↦ 7470
  Membership.mem ↦ 1560
  Set.Mem ↦ 1546
  LE.le ↦ 1437
  Subset ↦ 1423
  Set.Subset ↦ 1396
  setOf ↦ 762
  closure ↦ 90
  sInf ↦ 90
  sInter ↦ 90
  Set.Nonempty ↦ 22
[reduction] unfolded instances (max: 8158, num: 8): ▼
  PositiveCompacts.instSetLike ↦ 8158
  instAddNat ↦ 7374
  instHAdd ↦ 7374
  instOfNatNat ↦ 5531
  instMembership ↦ 1546
  instLE ↦ 1417
  instHasSubset ↦ 1407
  instInfSet ↦ 90
[reduction] unfolded reducible declarations (max: 46136, num: 4):
  PositiveCompacts.toCompacts ↦ 46136
```

Something like a unification trace tree can still be hard to survey; the new `diagnostics` option provides a nicer flat view for selected components such as declarations by number of unfoldings during unification. In Lean 4.9.0, definitions that really should not be unfolded in a specific context can be marked so using the `seal` command.

Summary

New tools for profiling in the large and in the small, visually and textually

Suggestions for what we’re still missing?