Profiling Tools in Lean

Sebastian Ullrich, Lean FRO
LMU, 2024/06/06
The Basics: **lean --profile**

```
$ lean --profile
```

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 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
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: \texttt{trace.profiler}

A very simple change turned out to provide much better data: the \texttt{trace.profiler} option takes the existing trace tree system used for debugging Lean and automatically activates and annotates any nodes above \texttt{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.
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.
 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.
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.
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?