Lean 4 for Software Engineering

Alok Singh
`alokbeniwal@gmail.com`

Abstract

Lean 4 combines programming and theorem proving. Unlike typical theorem provers, it is fast enough for practical software engineering. Its `sorry` construct enables incremental verification—developers can specify properties now and prove them later. Through case studies, I show how this approach allows moving fast without breaking things. Lean has a strong community and virtuous cycle with AI, and its synergy with the mathematics community ensures that it won't fade away. The result: practical software that's both fast and correct.

Contents

1 Introduction: Two Worlds

Lean 4 is a programming language that is also a theorem prover. It straddles the worlds of mathematics and software engineering.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

The combination of the two is notable because there are *many* general-purpose programming languages and several theorem provers. But few are both. And none which have the unified community of Lean.

Also (almost) uniquely among languages in its niche, Lean is implemented in itself. Agda is implemented in Haskell and lacks a robust tactic framework, Coq is implemented in Gallina/OCaml but is unsuitable for general-purpose programming, and Isabelle is implemented in Standard ML and is also unsuitable for general-purpose programming. Idris**?** is implemented in itself, but its theorem proving capabilities are much less developed. F**?** is the only other language that is both a programming language and a theorem prover, but its community is far smaller than Lean's.

Lean is fast. It runs at 20–50% of C++'s speed, much faster than Python, Coq, and Agda.. *Anecdotally*, many people I have encountered, most of whom are very experienced with programming languages, implicitly expected Lean to be slow–100x slower than C. They were very surprised to find that's not the case.

Also uniquely among the languages listed, Lean has many niceties for software engineering, similar to Rust. Like Rust's `cargo`, Lean has a builtin package manager and build tool, `Lake` , a package repository like `crates.io`**?** (Reservoir), version manager (Elan **?**), documentation tool (Lake) , and extensible linter with informative error messages. The success of Cargo and more recently in Python, `uv` **?** have shown that developers really prefer one-stop-shops for such needs. Non-software engineers especially appreciate this tooling, because it lowers the barrier to entry and encourages people to actually use the tools.

1.1 Why does this matter?

This quote from one of the creators of Isabelle illustrates why the combination of theorem proving and programming matters:

> But university departments are not software houses. Programs like Isabelle are not products: when they have served their purpose, they are discarded.
> Lawrence C. Paulson, "Isabelle: The Next 700 Theorem Provers"

The issue is that a theorem prover *is* a piece of software, and software grows over time. There is a lot of overlap between the needs of a theorem prover and a general-purpose programming language, and being good at both unlocks many benefits. A language implemented in itself avoids the 2-language problem, and lets less experienced users contribute to "advanced" software like the compiler without having to learn something extra (like C for contributing to the Python compiler).

Lean's software needs act as a rising tide for the whole language and ordinary programming. Its synergy with the mathematics community ensures that it won't fade away like some other niche languages.

2 The Lean Community

While this paper is about Lean for software engineering rather than math, we must talk about math because Lean's success is due to its core community: mathematicians.

Lean has captured the attention of the math community and gone past an inflection point to achieve a critical mass of users. Anecdotally, this inflection point was sometime in late 2023/early 2024, helped by Terence Tao using Lean. Despite being much younger than Agda (to take just one example), Lean has surpassed it in terms of users. And its development velocity is still increasing.

2.1 Breaking the Lisp Curse

This unity gives Lean a chance to break the Lisp Curse for software engineering.

The Lisp Curse is a phenomenon where powerful (meta)programming capabilities lead to fragmentation - everyone writes their own solution rather than collaborating on a shared one. This creates a tragedy of the commons where many half-finished solutions exist but none are production-ready. The name stems from Lisp's flexibility: it's so easy to write code that there's little incentive to reuse or polish existing solutions.

As Mark Tarver provocatively put it:

> One of these is the inability to finish things off properly... Lisp allows you to just chuck things off so easily, and it is easy to take this for granted. I saw this 10 years ago when looking for a GUI to my Lisp... No problem, there were 9 different offerings. The trouble was that none of the 9 were properly documented and none were bug free. Basically each person had implemented his own solution and it worked for him so that was fine...
>
> Now in contrast, the C/C++ approach is quite different. It's so damn hard to do anything with tweezers and glue that anything significant you do will be a real achievement. You want to document it. Also you're liable to need help in any C project of significant size; so you're liable to be social and work with others. You need to, just to get somewhere.

The task of formalizing mathematics is "so damn hard" that it has united the community. And this is not a theoretical claim, this has already happened. mathlib is by far the dominant library in Lean, though there are many smaller projects. But they operate with the support of the community, and the bazaar upstreams finished work to mathlib's cathedral.

Whether Lean can escape this for the software engineering side is less clear. But its mathematician userbase has so far also united around common software needs. As the community grows by attracting both mathematicians and software engineers, it has leadership to turn to.

Speaking of leadership:

2.2 Notable Contributors

2.2.1 Peter Scholze

Peter Scholze's Liquid Tensor project is an example of a large-scale effort that has contributions from non-experts. It formalized perfectoid spaces and the necessary commutative algebra, before proving a key lemma in Scholze's work. This is a very deep topic which only Scholze himself really had full knowledge of, yet the work was mostly done by others under his guidance. According to Kevin Buzzard**?**, this was the first project that really drew attention from more mainstream mathematicians, since it formalized a concept that they didn't already know.

2.2.2 Terry Tao

Terry Tao's breadth of work has brought attention to Lean, and may have been its tipping point. Tao is famous for wide collaboration, such as the Polymath project**?**. However, he has also expressed that Polymath's problem was that incorporating contributions from non-experts was difficult.

Tao has led the work on formalizing the The Polynomial Freiman-Ruzsa conjecture, a major problem in additive combinatorics solved by Tao and others in 2023. It was formalized in Lean within the same month, with most of the contributions coming from others.

Here is the state of the proof just 5 days after Tao released the preprint. Green is formalized, blue is "ready to be formalized" (the fringe of the dependency graph), white is not started. The rest was done within 2 weeks, much exceeding Tao's expectations.

His projects now attract 50+ contributors, exceeding Polymath's 40 or so.

This matters because Tao and Scholze are perhaps the most famous mathematicians in the world, and their involvement has brought attention to Lean.

3 Case Studies

3.1 lean-inf

`lean-inf` is an implementation of the Levi-Civita field**??**.

It allows computing expressions like $\epsilon^2 H^3 + \epsilon H^2 + 1$ where ϵ is an infinitesimal and H is its reciprocal, an infinite number, useful for automatic differentiation.

A note on syntax: Function calls are written as `f x` instead of `f(x)`, like in shell scripts or Haskell. `fun` is an anonymous function, and `fun input1 input2 => output` is a function that take 2 arguments `input1, input2` and returns `output`, equivalent to `lambda input1, input2: output` in Python.

Consider the following `structure` (like a `struct` in C/Rust or a `class` in Python). It has five fields: three for data and two for assumptions/"proofs":

```
def Coeff := Float
def Order := Rat -- rational number data type

structure LeviCivitaNumber where
  standard : Coeff
  infinitesimal : HashMap Order Coeff -- polynomial in H
  infinite : HashMap Order Coeff
  /-- Ensures all infinitesimal terms have negative order. -/
  _pf_infinitesimal_keys_negative : infinitesimal.all (fun order _ => order < 0) := sorry
  /-- Ensures all infinite terms have positive order. -/
  _pf_infinite_keys_positive : infinite.all (fun order _ => order > 0) := sorry
```

The data structure assumes that H is the basic "unit", and that all other terms are in terms of H. So H has order 1, standard finite numbers have order 0, and all infinite terms have positive order, and all infinitesimal terms have negative order. The last 2 fields ensure this assumption by encoding it directly into the type system. In other words, you can write your spec in code along with its implementation, and the type checker will verify that the implementation satisfies the spec.

A good comparison is Rust's borrow checker: it will not let you mess up ownership rules.

But Lean's verifier is much more sophisticated, since it can check not just ownership, but essentially anything. In a sense, the Rust borrow checker is a verifier for one property (ownership), and the Lean type checker is a verifier for *any* property.

3.1.1 Where did it go Wrong? Line and Column Numbers for Assumptions

Here's the equivalent code for the Levi-Civita field in Python:

```
Coeff = float
Order = fractions.Fraction # import `fractions`

@dataclass
class LeviCivitaNumber:
    standard: Coeff
    # XXX: Orders MUST be negative <- state of the art in most languages
    infinitesimal: dict[Order, Coeff]
    # XXX: Orders MUST be positive
    infinite: dict[Order, Coeff]
```

In most languages, such `XXX` comments are roughly the best we may do. This illustrates a benefit of the "prover side" for software engineering: assumptions become explicit, to the point of having line and column numbers to point out the exact place where an assumption is made (definition site) or satisfied/violated (use site). And Lean will not let such assumptions slip by without at least a compiler warning that `sorry` was used: they will not compile.

If something went wrong in the Python code, the answer to "Where did it go wrong?" may not even be well-defined since the comments are not part of the formal code.

3.2 `sorry`-friendly programming

Wouldn't proving this order positivity assumption each time be a pain? Yes, it often is. But that's what the `:= sorry` is for. It is a default value for the field (a proof of the property). And `sorry` is a placeholder, which allows a program to typecheck without having to figure out the proof.

`sorry` is what lifts proving from an esoteric feature to a practical one: it lets you write the code without having to figure out the proof, but you gain the advantage of assumptions becoming explicit code. It looks a bit tacky to write `sorry` everywhere, but there's much uglier code out there in the wild. And you can always fill it in later with actual proofs.

And if even that is too much work, you just write test cases like in any other language. So you get the best of both worlds: the safety of a theorem prover, and the flexibility of a general purpose programming language, with graceful degradation to ordinary software engineering practices when needed.

3.2.1 Code Review, AI, and `sorry`

Consider `SciLean`. It is a scientific computing assistant that is like a hybrid between Jax and a computer algebra system (CAS), but with proofs used to construct executable code, similar in flavor to compiler optimizations. Its creator, Tomas Skrivan, is a proponent of "sorry-friendly programming", so `SciLean` has over 200 `sorry`s in 147 files as of this writing.

The authors of LeanAgent recently submitted a pull request to `SciLean` that fills in `sorry`s across 15 files. Many are simple proofs, but several use library-specific knowledge, and most importantly they were proved without human effort. We will return to this point in the next section.

This shows how such a tool changes the nature of code review: if the definitions are correct, and the code compiles, then the code is proved to be correct.

Normally code review is a tedious process of reading through code, and checking that it does what it's supposed to do, before actually reviewing its readability, style, performance, etc. But here the whole checking step is automated, so the reviewer can focus purely on the quality of the code.

3.3 `llm.lean`: A case study in AI assistance

Here is a more involved snippet, modified from llm.lean. It defines a function that computes the derivative of the attention mechanism, key to the Transformer model. The implementation of it took less than a minute thanks to the interaction between AI and Lean's verifier.

```
/-- No more shape errors! The compiler checks it for us.  -/
structure Vector (len: Nat)  where
  /-- Underlying data. -/
  data: Array Float
  /-- a proof that `data.length = len`. -/
  sizeIsRight: data.size = len := sorry

/-- A dense layer without bias. -/
structure DenseNoBias (Rows: Nat) (Cols: Nat) where
  weights : Vector Cols (Vector Rows)

def attention_backwards
  (dout: Vector T (Vector D))
  (q: Vector T (Vector D))
  (k: Vector T (Vector D))
  (v: Vector T (Vector D))
  -- returns tuple of (dq, dk, dv)
  : (Vector T (Vector D)) × (Vector T (Vector D)) × (Vector T (Vector D)) :=
  let a := q * k.transpose
  let norm_factor :=  1 / (D.toFloat).sqrt
  let a1 := a.map (fun x => x.map (fun y => y * norm_factor)) -- normalize
  let a2 := a1 + tril (-Float.inf) -- mask
  let a3 := a2.map softmax
  let dv := a3.transpose * dout
  let da3 := dout * v.transpose
  let da2 := Vector.zipWith softmax_backward da3 a3 -- softmax derivative
  let da1 := da2.map (fun x => x.map (fun y => y * norm_factor)) -- normalize
  let (dq, dk) := (da1 * q, da1 * k)
  (dq, dk, dv)
```

The following is all that we needed to write:

```
def attention_backwards
  (dout: Vector T (Vector D))
  (q: Vector T (Vector D))
  (k: Vector T (Vector D))
  (v: Vector T (Vector D))
  -- (dq, dk, dv)--^cursor is here
```

Copilot immediately suggested the rest of the code. However, it had 2 bugs, isolated in the following code:

```
let dv := a3 * dout.transpose
let da3 := dout * v
```

`dv` had a transpose in the wrong place, and `da3` had none. However, the type checker gave shape errors within milliseconds, and Copilot immediately suggested the fix before we could even read the error message. One more tab keypress, and it was done.

A major problem in modern AI is correctness. This hurts scalability. Deep chains of reasoning are hard, even for humans. The glut of CVEs for C/C++ is a (grim) testament to this. By Micosoft's estimation, $\tilde{7}$0% of their CVEs were memory-safety related.**?**

But Lean's verifier creates a virtuous cycle with AI tools: they can use their flexible intuition to write code, and the verifier can, well, *verify* it. Feeding back errors lets the AI fix its own mistakes.

The asymmetry in difficulty between specification and implementation allows people to write down relevant properties (or let the AI do it), while AI can provide flexible intuition to implement them. The verifier ties it all together, creating a virtuous cycle where AI-generated code is verified by Lean, and errors are systematically identified and corrected.

4 Connection to Singular Learning Theory and AI Safety

Tegmark and Omohundro have argued that formal verification is the only way to fully ensure AI safety **?**.

To do such verification, existing provers are a nonstarter. Coq is too slow and has no numerical support. It must be a programming language that is also a prover, and as argued above, the *only* such language is Lean.

Python is also unsuitable, since it is not verified, and retrofitting verification would be unfeasible.

Full verification is probably impractical, but partial verification is still useful, and fields like mechanistic interpretability and singular learning theory are working on related concerns. But the numerics can be brought in line with symbolic verification, and if work is done in formal verification for safety, it's difficult to imagine it being done in something besides Lean given the arguments above.

5 Conclusion

The apex predator in software is complexity, and Lean's formal verification helps manage complexity by making assumptions explicit and checkable, and providing powerful tools to do so. Its uniquely strong community and virtuous cycle with AI position Lean 4 well. Much work remains to be done, especially in writing more software libraries and pure numerical code, but projects like `SciLean` show that it is possible and natural to do so. The sea is rising, and it may penetrate the hard marl.

Acknowledgments and Disclosure of Funding

I thank the Lean community for their hard work, Tomas Skrivan for introducing me to sorry-friendly programming, and Govind Pimpale for working on `llm.lean` with me.