How to Use Lean 4, linarith, and ring to Help You Check Your Algebra Homework Without Cheating

jum-p'éel • 29 Jun 2025

How to Use Lean 4, linarith, and ring to Help You Check Your Algebra Homework Without Cheating

In the age of calculators and computer algebra systems, students face a dilemma: how can they verify their algebraic work without simply having a machine solve the entire problem for them? Lean 4, a modern theorem prover, offers an elegant solution. By using Lean's `linarith` and `ring` tactics strategically, students can check each step of their algebraic reasoning while still doing the mathematical thinking themselves.

The Problem with Traditional Tools

Most computer algebra systems like Wolfram Alpha or graphing calculators work by giving you the final answer immediately. Enter "solve 3x + 6 = 21" and you get "x = 5" without any insight into the process (to be fair, you can ask Wolfram Alpha for the steps after you got the answer, but that is too late). This creates a dependency where students either avoid computer help entirely (and potentially make uncaught errors) or rely too heavily on machines (and miss the learning opportunity).

Lean 4 changes this dynamic by serving as an "algebraic referee" rather than an "algebraic solver." It can verify that each step you take is mathematically valid without telling you what the next step should be.

Setting Up Your Verification Environment

To use Lean 4 for step-by-step verification, you need minimal setup:

```
import Mathlib.Data.Real.Basic
import Mathlib.Tactic
```

```
example (x : ℝ) (h : 3 * x + 6 = 21) : True := by
  -- Your work goes here
  sorry
```

This creates a sandbox where you can work with real numbers and test your algebraic manipulations. The goal `True` is deliberately trivial - we're not trying to prove anything specific, just explore what happens at each step.

Linear Algebra: The Power of linarith

For linear equations and inequalities, `linarith` becomes your verification tool. Instead of asking "what is x?", you make claims about what you think the next step should be, and `linarith` confirms whether your reasoning is correct.

Consider solving 3x + 6 = 21:

```
import Mathlib.Data.Real.Basic
import Mathlib.Tactic
example (x : ℝ) (h : 3 * x + 6 = 21) : True := by
    -- Step 1: I think subtracting 6 from both sides
        gives 3x = 15
    have step1 : 3 * x = 21 - 6 := by linarith [h]
    -- Step 2: I think 21 - 6 equals 15
    have step2 : 21 - 6 = 15 := by norm_num
    -- Step 3: So 3x = 15
    have step3 : 3 * x = 15 := by linarith [step1,
        step2]
    -- Step 4: I think x = 15/3 = 5
    have step4 : x = 15 / 3 := by
      have h_nonzero : (3 : ℝ) ≠ 0 := by norm_num
      linarith [step3, h_nonzero]
    sorry
```

Notice what's happening here: you decide what each transformation should be, and `linarith` verifies that it follows logically from the previous steps. If you make an error - say, claiming that subtracting 6 gives you 3x = 14 - Lean will reject that step with an error message.

Polynomial Manipulation: Enter ring

For more complex algebraic expressions involving polynomials, the `ring` tactic handles verification. This is particularly useful for expanding expressions, factoring, or checking polynomial identities.

```
import Mathlib.Data.Real.Basic
import Mathlib.Tactic

example (x : ℝ) : True := by
  -- Check: does (x + 2)² expand to x² + 4x + 4?
  have expansion : (x + 2)^2 = x^2 + 4*x + 4 := by rin
        g
  -- Check: does x² · x³ equal x⁵?
  have exponent_rule : x^2 * x^3 = x^5 := by ring
  -- Check: does (2x)(3x) equal 6x²?
  have multiplication : (2*x) * (3*x) = 6*x^2 := by ri
        ng
  -- Since we're proving True, we can use trivial
  sorry  -- Since we're proving True, we could use
        trivial too
```

The beauty of `ring` is that it verifies polynomial identities instantly. You can test your factorizations, expansions, and simplifications without having to work through all the tedious arithmetic by hand, but you still need to know what the result should be.

Working Step-by-Step: A Complete Example

Let's see how this works with a more complex problem. Suppose you're asked to solve $x^2 - 5x + 6 = 0$ by factoring:

```
import Mathlib.Data.Real.Basic
import Mathlib.Tactic

example (x : ℝ) (h : x^2 - 5*x + 6 = 0) : True := by
  -- Step 1: I think this factors as (x-2)(x-3) = 0
  -- Let me verify that (x-2)(x-3) expands to x²-5x+6
  have factorization : (x - 2) * (x - 3) = x^2 - 5*x +
        6 := by ring
  -- Step 2: So my original equation becomes (x-2)
        (x-3) = 0
  have factored_form : (x - 2) * (x - 3) = 0 := by
```

```
    rw [factorization]
    exact h

        -- Step 3: By zero product property, either
        x-2=0 or x-3=0
  -- This means x=2 or x=3
  sorry  -- Since we're proving True, we could use
        trivial
```

Here, you had to figure out the correct factorization yourself (the hard part!), but Lean verified that your factorization was algebraically correct and that it was equivalent to the original equation.

The Learning Benefits

This approach offers several pedagogical advantages:

Active Learning: You must think through each step rather than passively receiving an answer. The cognitive load of deciding what to try next keeps you engaged with the mathematical reasoning.

Immediate Feedback: Unlike working problems on paper where errors might not surface until you check your final answer, Lean catches mistakes at each step. This prevents you from building incorrect reasoning on top of early errors.

Confidence Building: When Lean accepts your step, you know it's mathematically sound. This builds confidence in your algebraic skills and helps you recognize correct reasoning patterns.

Error Isolation: If you make a mistake, you know exactly where it occurred. Instead of having to re-solve the entire problem, you can focus on understanding why that particular step was invalid.

Avoiding the Cheating Trap

The key to using Lean ethically for homework is maintaining the right relationship with the tool. Lean should verify your thinking, not replace it. Here are some guidelines:

- **Don't use Lean to generate solutions:** Resist the temptation to try random manipulations until something works.
- **Plan your steps first:** Think through your approach on paper before encoding it in Lean.

- **Understand rejections:** When Lean rejects a step, work to understand why rather than just trying alternatives.
- **Use it for verification, not exploration:** Lean works best when you have a clear idea of what you want to check.

Limitations and Realistic Expectations

Lean isn't magic. It has limitations that are important to understand:

- **Setup overhead:** Getting the imports and syntax right takes time initially.
- **Syntax learning curve:** Lean's notation may differ from standard mathematical notation.
- **Limited scope:** Some advanced techniques (like trigonometric identities) require additional libraries or tactics.
- **Error messages:** When things go wrong, Lean's error messages can be cryptic for beginners.

Conclusion

Lean 4 with `linarith` and `ring` offers a middle path between doing algebra completely by hand and having a computer solve everything for you. It preserves the essential learning experience - the mathematical reasoning and decision-making - while providing reliable verification of each step.

This approach treats mathematics as a dialogue between human intuition and mechanical verification. You provide the mathematical insight and creative problem-solving, while Lean ensures logical consistency and computational accuracy. The result is a learning experience that builds both confidence and competence in algebraic manipulation.

For students willing to invest in learning the basic Lean syntax, this method can transform algebra homework from a source of anxiety into an opportunity for mathematical exploration with a safety net. You get to be the mathematician, while Lean serves as your meticulous, patient, and always-available teaching assistant.