A Lean Tactic for Normalizing Expressions in an Algebra over a Ring

Arend Mellendijk
University of Bonn, Germany

Abstract

This paper introduces the `algebra` normalizing tactic for the Lean theorem prover. This tactic expands on the existing `ring` tactic by additionally supporting a scalar multiplication action over a fixed commutative (semi)ring. It supports rational constants in the base ring even when the main ring is not a field, which lets us implement a suite of tactics for manipulating both univariate and multivariate polynomials. These features are implemented by adapting the existing implementation of `ring` while retaining support for variable exponents.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms; Theory of computation → Automated reasoning; Theory of computation → Type theory

Keywords and phrases Lean, Mathlib, algebraic structures, proof algorithms

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.35

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/amellendijk/algebra-tactic` [10], archived at `swh:1:dir:c939acdbc385233836c284ca55f264661c444c36`

Declaration about GenAI/LLM use Claude Code was occasionally used as an assistant in writing the tactic described in this paper. The statements in `Algebra/Lemmas.lean` were created by Claude using type information provided by the Lean LSP. Its use was otherwise limited to cleaning up existing code. All changes were carefully reviewed by the author.

1 Introduction

In the formalization of mathematics, we often employ polynomial-like structures, be it univariate or multivariate polynomials, rational functions or formal power series. For this reason, it is imperative that we have good automation tools for manipulating them. What makes a structure A polynomial-like is that it is a commutative (semi)ring with clearly defined coefficient ring R and a preferred ring homomorphism $R \to A$. This data makes A into an R-algebra.

The `ring` tactic in Lean's [3] mathematical library `Mathlib` [9] proves equations in a commutative (semi)ring. Its implementation in Lean 4 is based on Baanen's `ring_exp` tactic from Lean 3, described in [1]. Similar tactics exist in other proof assistants [5, 6, 11, 12]. The reason `ring` is not sufficient for proving equations in polynomial-like structures, is that `ring` does not normalize expressions in both the coefficient ring and the polynomial ring at the same time.

To be more specific, `ring` does not support ring homomorphisms (such as the inclusion map $R \to R[X]$) in its core normalization procedure. Typically, users will apply the `norm_cast`/`push_cast` tactics [7] or call the simplifier with appropriate lemmas in order to "push" the homomorphisms through addition, multiplication and constants before calling `ring`. Such a preprocessing step is part of `math-comp`'s [8] implementation of `ring` [12].

This approach has a major downside. Consider the inclusion map $C : \mathbb{Q} \to \mathbb{Q}[X]$. The above approach would fail to prove $C(1/3) + C(2/3) = 1$. Because the codomain of C is not a field, we have no good way of pushing C through division in a way that Lean's numerical

normalization procedures can handle. In this example, it would suffice to instead "pull" C back and compute the sum in $\mathbb{Q}$ instead, but in more complicated expressions other terms will get in the way. Our `polynomial` tactic can prove this example directly.

Of course, one also has the inverse situation where one has a field F which is an algebra over a ring R (a number field over its ring of integers, say), where rational constants can be expressed in F but not in R. Our procedure makes no effort to normalize rational constants in this setting, and indeed this is the setting where the existing strategy of "pushing" homomorphisms and then calling `ring` suffices. We will note that this setting also never occurs in the case of (multivariate) polynomials over their coefficient ring.

We will first describe a modification of `ring`'s normalization procedure which allows for a scalar multiplication action induced by an `Algebra` instance, and then use this new procedure to develop a suite of tactics for proving equations in either the general setting of a commutative algebra over a ring, or the specific setting of a polynomial-like structure. The following examples show this suite in action. The supplementary materials contain a number of additional examples of theorems these tactics can prove.

Example 1 Our tactic suite in action.

```
/-- ‘algebra‘ supports variable exponents in both the base ring and the main ring.
It also supports rationals in the base ring when the main ring is not a field. -/
example {R A : Type*} [Field R] [CharZero R] [CommRing A] [Algebra R A]
    (n : ℕ) (r : R) (a : A) :
    (1 + (1/2 * r ^ n) • a) ^ 2 = 1 + (r ^ n) • a + (1/4 * r ^ (2*n)) • a^2 := by
  algebra

/-- ‘algebra_nf‘ groups coefficients in the base ring by the atoms they multiply -/
example {R A : Type*} {a b : R} [CommSemiring R] [CommSemiring A] [Algebra R A]
    (x y : A) : ((a + b) • (x + y))^2 = 0 := by
  algebra_nf with R
  /- The goal is now :
    (a * b * 4 + a ^ 2 * 2 + b ^ 2 * 2) • (x * y) +
    (a * b * 2 + a ^ 2 + b ^ 2) • x ^ 2 +
    (a * b * 2 + a ^ 2 + b ^ 2) • y ^ 2 = 0 -/
```

Unlike in `math-comp` [12], neither `ring` nor `algebra` use reflection. This is standard for tactics in `Mathlib`. We manipulate normalized expressions as `Expr`s at the meta level, and the normalization procedure constructs a large proof term that is type checked by Lean's kernel. Lean does have a reflection based normalization tactic for commutative (semi)rings, which is built into the `grind` tactic. `grind`'s tactic does not support variable exponents.

2 The Normalization Procedure

We will start with a brief overview of `ring`'s normalization procedure. `Mathlib`'s `ring` does not use the Horner normal form. Instead, the normal form is stored roughly as a list of multivariate monomials that supports variable natural number exponents, which are themselves kept in normal form.

The grammar of the normal form is defined using three inductive types called `ExSum`, `ExProd` and `ExBase`. An `ExBase` is an expression that cannot be normalized because its outermost operation is not an addition, multiplication, or another operation supported by `ring`. We call such an expression an *atom*. An `ExProd` is a product of `ExBase`s, each raised to some power, and terminated by a constant. An `ExSum` is a sum of `ExProd`s. We will refer to these types collectively as the `Ex` types.

To normalize an expression $x + y$, `ring` first normalizes x and y into `ExSums`. It then evaluates the sum of the two `ExSums` by combining like terms and maintaining various invariants about the order of the monomials. That is, to evaluate addition, `ring` implements a function that takes two normalized expressions x' and y', and produces a third normalized expression z alongside a proof that $x' + y' = z$.

The constant coefficients are kept track of internally using rational numbers. Numerical computations on the constant coefficients are done using Lean's `norm_num` tactic. `norm_num` uses the minimal necessary typeclass assumptions to express the rational number in a given type. In particular, if our ring is a field of characteristic zero then `ring` supports rational constants, even though `ring` itself does not have a normalization procedure for dividing by a non-constant `ExSum`.

Algebra

`algebra`'s normalization procedure starts with rings R and A with `CommSemiring` instances on them, as well as an `Algebra R A` instance. Since our rings are commutative, the algebra instance simply stores the data of a favoured ring homomorphism called `algebraMap R A : R →+* A`, which induces a scalar multiplication action by `r • a = algebraMap R A r * a`. This compatibility condition follows from our typeclass assumption as described in [13]. As described later in this section, we will attempt to choose R to be as general as possible, in the sense that any other ring R' with an `Algebra R' A` instance can be cast into R.

`algebra`'s normal form is similar to `ring`'s. For clarity we will distinguish between the two `Ex` types by prepending them with either `Ring` or `Algebra`.

In order to support the scalar action of R on A, we define `Algebra.ExProd` by modifying `Ring.ExProd` by attaching a coefficient r in R. This coefficient will be kept in `ring` normal form, and is therefore stored as a `Ring.ExSum`. In fact, we don't bother storing a rational constant in `Algebra.ExProd` and we lift it directly to the coefficient in R. `Algebra.ExBase` and `Algebra.ExSum` are defined in the same way as `ring`, mutatis mutandis.

This idea of storing all constants in the R-coefficients is what allows us to support rational coefficients in polynomial-like rings, but if A is a field and R is not then we are unable to lift the constant into R. The constant is then treated as its own atom, and can no longer be used in any numeric computations. In this case, though, one can resort to using `push_cast` followed by `ring`, as there will be no rational constants for homomorphisms to get stuck on.

Addition, multiplication, subtraction, etc. are all implemented similarly to `ring`, except that operations on the coefficients are now evaluated using `ring` rather than `norm_num`. We additionally support scalar multiplication. To evaluate `(r : R) • a`, we normalize r using `ring` and a using `algebra`. Then we distribute the `Ring.ExSum` r over the `Algebra.ExSum` a by turning r into an `Algebra.ExProd` (that is, an empty product with r as a coefficient) and using the existing machinery for multiplication.

When we encounter a scalar multiplication $r' \bullet a$ over a ring R' different from R, we try to synthesize `Algebra R' R` and `IsScalarTower R' R A` instances. The `Algebra` instance gives us a map $R' \to R$, which we can use to lift r' to R. The `IsScalarTower` instance tells us that the new scalar multiplication action of R' on R is compatible with the existing actions of R' and R on A, in the sense that $(r' \bullet r) \bullet a = r' \bullet (r \bullet a)$. In particular, it tells us that $r' \bullet a = (\texttt{algebraMap R' R } r') \bullet a$. A more thorough description of the `IsScalarTower` typeclass can be found at [2]. We also then push the `algebraMap` as far into r as possible, using `push_cast`. If either of the typeclasses fail to synthesize, we simply treat $r' \bullet a$ as an atom.

Our procedure assumes both R and A are known before we start normalizing the expression. We can infer A from the type of our expression, but R requires inspecting the expression more closely. To determine what R should be, we collect a list of all rings that appear as the base ring in a scalar multiplication somewhere in the expression. If we find no such rings, we set R to be either $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}_{\geq 0}$ or $\mathbb{Q}$ depending on whether A is a semiring, ring, or (semi)field of characteristic zero, respectively. If there are multiple rings, we try to pick the "largest", in the sense that R_1 is larger than R_2 if R_1 is an algebra over R_2. This is a natural choice, as it lets us lift as many of the scalar multiplications as possible into R. There might not be a well-defined largest ring. For this reason, we always give the user the ability to override this choice by providing their own ring.

3 Implementation Details

In order to minimize code duplication between `ring` and `algebra`, we first generalized the `Ex` types that describe `ring`'s normalization procedure so that the normal forms of the coefficients can be stored in a custom type `BaseType`. The `BaseType` for `ring` simply stores a rational number and, if it is not an integer, a proof that the denominator is nonzero in our ring. For `algebra` the `BaseType` is an `ExSum` in R, so that the coefficients are terms in R that are kept in `ring` normal form.

We then defined an interface we call `RingCompute` that specifies the precise operations that must be supported on a `BaseType` for the normalization procedure to function. `Ring` implements the interface using the internal API for `norm_num`, while `algebra` uses `ring`'s internal normalization functions.

Finally, all calls to `norm_num` were extracted and replaced with the generic interface. Some care had to be taken to handle exponents, since they should always be normalized according to `ring`'s procedure, no matter the `BaseType`.

Snippet 2 shows part of the `RingCompute` interface. The `mul` function evaluates the product of two normalized expressions in the `BaseType`, returning a new normalized expression and a proof of correctness. The `Result` type here stores the normalized expression and a proof that it is equal to the product of the inputs.

The `add` function does essentially the same thing, but for addition. If the result is zero, it additionally returns a proof that `IsNat (x+y) 0`, so that the normalization algorithm can effectively cancel terms.

Snippet 2 The interface for a custom coefficient type.

```
structure RingCompute {u : Lean.Level} {α : Q(Type u)}
    (BaseType : Q($α) → Type) (sα : Q(CommSemiring $α)) where
  add {x y : Q($α)} : BaseType x → BaseType y →
    MetaM ((Result BaseType q($x + $y)) × (Option Q(IsNat ($x + $y) 0)))
  mul {x y : Q($α)} : BaseType x → BaseType y →
    MetaM (Result BaseType q($x * $y))
  /- Other entries omitted. -/
  ...
```

This snippet uses the `Qq` package [4] to provide type annotations to Lean's meta-level `Expr` type. `x : Q($α)` indicates that `x` is an expression of type α, and `q($x + $y)` is an expression representing $x + y$ constructed using the type hints given to `Qq`.

The argument `x` in `BaseType x` is the normalized expression in α whose structure is represented by `BaseType x`. We pass `x` as a parameter rather than storing it in the structure, because it lets us add type annotations to our functions and inductive types. In practice, we rarely have to build an expression manually besides proof terms, as they can usually be inferred from the type annotations.

4 The Tactics

We used this normalization procedure to implement six user-facing tactics. Three of them operate in a general commutative algebra over a commutative ring; and the remaining three are variants specialized to polynomials with coefficients in a commutative ring.

`algebra`

The `algebra` tactic is perhaps the simplest to describe. It checks that we are currently proving an equality, normalizes both sides and closes the goal if both normal forms are equal.

`algebra_nf`

Inspired by `ring_nf`, `algebra_nf` takes a list of local hypotheses (or the goal), and normalizes all of their subexpressions. When evaluating an atomic expression, it first recursively normalizes using `algebra_nf` so that e.g. e^{xy} and e^{yx} evaluate to the same atom.

We navigate the expression by using the simplifier with a custom simplification procedure that normalizes an expression if its type is a commutative semiring. We want to avoid it inferring the base ring independently on each subexpression, leading to different normal forms on what are morally the same expression. For example if $n \in \mathbb{N}$ and $x \in \mathbb{Q}$ then the equation `n • x + x = (n : ℤ) • x + x` normalizes to `(n + 1 : ℕ) • x = (n + 1 : ℤ) • x`; while `algebra` would close the goal as it uses $\mathbb{Z}$ as the base ring on both sides of the equation. To remedy this, we tell the user to provide an explicit base ring by providing a code action urging them to replace `algebra_nf` by `algebra_nf with R`. Here we attempt to infer R from the base rings appearing in all subexpressions.

Example 3 `algebra_nf` normalizes inside subexpressions.

```
example {R A : Type*} [Field R] [CharZero R] [CommSemiring A] [Algebra R A]
    {P : A → Prop} {a : A} (h : P 0) :
    P (((2/3 : R) · a)^2 + (-4/9 : R) · a^2) := by
  algebra_nf with R
  exact h
```

The output of the normalization procedure by itself is not pretty. The products are associated the wrong way around, the constants are not expressed in standard form, and the lifting mechanism uses `algebraMap ℕ R` instead of `Nat.cast`. These issues are cleaned up during a postprocessing step using the simplifier with a custom set of lemmas.

`match_scalars_alg`

`match_scalars` is a tactic in `Mathlib` for proving equations in a module M over a (possibly noncommutative) semiring R. It parses both sides of an equation as an R-linear combination of M-atoms, and then matches identical M-atoms to produce proof obligations showing that

their coefficients are equal. If $m_1, m_2 \in M$ and $r_1, r_2 \in R$ then a goal $r_1 \bullet m_1 + m_2 + m_1 = r_2 \bullet m_1 + m_2$ would leave a single side goal that $r_1 + 1 = r_2$. Note that this tactic does not make use of a commutative multiplication operation on M, so that $a \bullet (x * y) = a \bullet (y * x)$ produces two side goals of $a = 0$ and $0 = a$.

`match_scalars_alg` removes this limitation. It parses both sides of an equation according to `algebra`'s normalization procedure, after which it walks through both sums and matches identical monomials in A to create side conditions showing that their coefficients in R are equal. Since these coefficients are kept in `ring` normal form, it also runs the `ring_nf` postprocessor on each side condition.

```
example (a b c : ℤ) (x : ℚ) : (1 + a • x)^2 = 1 + c • x + b • (x ^ 2) := by
  match_scalars_alg
  /- Remaining goals: ⊢ a ^ 2 = b, ⊢ a * 2 = c -/
```

Polynomials

Given a commutative (semi)ring R, the (semi)ring of polynomials $R[X]$ is an algebra over R with the natural inclusion map. One would expect the previous three tactics described above to apply directly. But in `Mathlib` it is standard to express the ring homomorphism $R \to R[X]$ using `Polynomial.C` and not `algebraMap R R[X]`. Indeed, `Mathlib` cannot replace `Polynomial.C` by `algebraMap` entirely, since the `Algebra R R[X]` instance assumes that R is commutative and `Polynomial.C` does not.

But polynomials are an essential use case. To handle polynomials we develop three variants of the previous tactics, namely `polynomial`, `polynomial_nf` and `match_coefficients`. These all differ in two ways.

As a preprocessing step, we run the simplifier with a custom set of lemmas to remove type-specific idioms that can be rephrased in terms of `algebraMap`s. For polynomials, these lemmas look like `C = algebraMap R R[X]` and `monomial n r = algebraMap R R[X] r * X ^ n`.

The appropriate base ring for normalizing an expression r in $R[X]$ is R. The base ring can be inferred from the type of r, so we do not need to collect the base rings that appear in r itself to determine R. This yields a much more robust inference mechanism than `algebra`, and a more well-defined normal form for `polynomial_nf`

For `polynomial_nf` specifically, we also have to handle cleanup. The output of the normalization procedure is phrased in terms of `algebraMap`s, which would be nonstandard for polynomials. To remedy this, we again run a simplifier with custom lemmas as a cleanup procedure.

Example 4 `polynomial_nf` produces an idiomatic expression.

```
example (a : ℚ) (n : ℕ) : (C a * X ^ n + C (1/3))^2 = 0 := by
  polynomial_nf
  /- Goal: C (1 / 9) + C (a * (2 / 3)) * X ^ n + C (a ^ 2) * X ^ (n * 2) = 0 -/
```

Univariate polynomials are not the only type with this issue. Multivariate polynomials and formal power series suffer from the same problem. Similarly, a user might define their own computable polynomial type (indeed, `Mathlib`'s polynomials are marked noncomputable.)

For this reason, we've taken care to ensure that the `polynomial` tactics are extensible by the user. To add support for a new type, one must:

- Tag preprocessing lemmas with `@[polynomial_pre]`
- Tag cleanup lemmas with `@[polynomial_post]`
- Register an environment extension that attempts to infer the base type from an expression

5 Discussion

We generalized the ring tactic to allow for coefficients in a more general type than $\mathbb{Q}$. This work might itself be helpful in the future by letting `ring` support coefficients in more general computable types. One can imagine using a computable type of algebraic numbers, thus improving `ring`'s behaviour when there are constants involving n-th roots.

Instead of pulling rational constants into the base ring R, we could have attempted to cast rational constants directly into A using an internal cast function and pull the constants back into R during cleanup. We chose not to pursue this approach in large part because we were interested mainly in the polynomial case, where grouping coefficients by their monomial is generally preferable. Additionally, `norm_num` currently only represents rational constants in division rings, so we would have to reimplement the evaluation functions on the rational constants, keeping track of the fact that the denominators are units in A. Still, the generalization of `ring` to a more general coefficient type makes this alternate approach easier to implement, as it precisely isolates the problematic reliance on `norm_num`.

The `algebra` normalization algorithm is almost strictly stronger than `ring`. To test our tactic and appraise its performance, we created a macro that replaces all explicit occurrences of the `ring` tactic in `Mathlib` with `algebra`. We found that all but three of the replacements succeeded. In all three cases, the tactic inferred a base ring which did not support the necessary algebraic operations (subtraction or inversion), and the tactic could be made to succeed by manually choosing an appropriate base ring.

When compiling `Mathlib` with these changes, we found that a total of 16 seconds (`wall-clock` time) were spent running the `algebra` tactic, while the time spent running `ring` went down by 12 seconds from 36 to 24. This corresponds roughly to a 33% increase in compile time of `algebra` compared to `ring` on real-world problems. Note this number is rather imprecise, if only because our measurements are rounded to the nearest second. The time spent in `ring` did not go to zero because the test did not affect tactics calling `ring` as a subroutine, including `ring_nf`. These times were measured using profiling infrastructure provided by the Lean FRO.

The expressions that tend to appear in `Mathlib` are not very large. To get an idea for the performance on larger terms, we ran both `ring` and `algebra` on the target $1 + (x + y)^n = (y + x)^n + 1$ for $n = 10, 20, 40$ and counted the number of heartbeats each used as a hardware-independent proxy for performance. In these examples, we observe that `algebra` uses around 50% more heartbeats than `ring`. Given the performance degradation, we are hesitant to replace `ring` by `algebra` throughout `Mathlib`. In everyday use, however, `ring` is already considered very fast, and a 50% increase in compilation time would not be very noticeable.

The main `algebra` tactic is currently available in `Mathlib`.[1] We are currently working to merge the remaining tactics. Once they are merged, `Mathlib` users will have a much easier time working with polynomials.

[1] The tactic was added in the commit with SHA `105b370555a4386b81ebd6a80e65a6cabd69c455`

References

1. Anne Baanen. A Lean Tactic for Normalising Ring Expressions with Exponents (Short Paper). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 21–27, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-51054-1_2.
2. Anne Baanen, Sander R. Dahmen, Ashvni Narayanan, and Filippo A. E. Nuccio Mortarino Majno di Capriglio. A Formalization of Dedekind Domains and Class Groups of Global Fields. *Journal of Automated Reasoning*, 66(4):611–637, 2022. doi:10.1007/s10817-022-09644-0.
3. Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-21401-6_26.
4. Gabriel Ebner. Qq: Intuitive, type-safe expression quotations for Lean 4. URL: https://github.com/leanprover-community/quote4.
5. Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, pages 98–113, Berlin, Heidelberg, 2005. Springer. doi:10.1007/11541868_7.
6. John Harrison. HOL Light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, pages 265–269, Berlin, Heidelberg, 1996. Springer. doi:10.1007/BFb0031814.
7. Robert Y. Lewis and Paul Nicolas Madelaine. Simplifying casts and coercions (Extended Abstract): Joint of the 7th Workshop on Practical Aspects of Automated Reasoning and the 5th Satisfiability Checking and Symbolic Computation Workshop, PAAR+SC-Square 2020. *PAAR+SC-Square 2020 Practical Aspects of Automated Reasoning and Satisfiability Checking and Symbolic Computation Workshop 2020*, pages 53–62, 2020. URL: https://www.scopus.com/pages/publications/85097274168.
8. Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2022. Version Number: 1.0.2. doi:10.5281/zenodo.7118596.
9. The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.
10. Arend Mellendijk. algebra-tactic. Software, swhId: swh:1:dir:c939acdbc385233836c284ca55f264661c444c36 (visited on 2026-07-13). URL: https://github.com/amellendijk/algebra-tactic, doi:10.4230/artifacts.27124.
11. Tobias Nipkow, Markus Wenzel, Lawrence C. Paulson, Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen, editors. *Isabelle/HOL*, volume 2283 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2002. doi:10.1007/3-540-45949-9.
12. Kazuhiko Sakaguchi. Reflexive Tactics for Algebra, Revisited. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:22, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.29.
13. Eric Wieser. Scalar actions in Lean's mathlib, 2021. arXiv:2108.10700 [cs]. doi:10.48550/arXiv.2108.10700.