Automate Polynomial: An Experiment in Reflection for Polynomials

Liam Schilling Quang Dao

August 18, 2025

Abstract

Polynomials are crucial to cryptographic protocols for their error-checking applications. Proof assistants like Lean 4 enable machine-verified implementations of those protocols, yielding more correct and secure systems. While those implementations demand an efficient way to prove properties of polynomials in Lean, representations of polynomials in Lean's mathematics library are not directly computable, making simple results tedious to prove. To address this issue, we design and implement a general proof-by-reflection model in Lean, reducing mathematical problems to decisions on computable representations.

The resulting systems automate proof of degrees, coefficients, evaluations, and expansions for univariate and multivariate polynomials in various contexts. The model's design specifies three levels of abstraction, producing modular and reusable tactics that are generic to computable representations of properties. We demonstrate this by exploring representations such as lambda functions for evaluations, dense lists for univariate coefficients, and dense trees for multivariate coefficients. Sparse and array representations are priorities for future work to improve efficiency. This work provides automated proving tools for polynomials and a general proof-by-reflection model for Lean, contributing to the development of reliable, machine-verified systems.

1 Introduction

1.1 Organization of the Blueprint

The blueprint is organized as follows: The rest of Section 1 examines previous work in proof automation and polynomials in Lean, and provides a brief comparison with the approach taken in our work. Section 2 describes our approach to proof by reflection and outlines a model for its implementation. This section will also be of interest to readers curious about our use of type-class inference in proof automation. Section 3 details the implementation of each part of the model for univariate polynomials. Readers interested in using the univariate system for proof automation should be directed specifically to Section 3.4. Section 4 addresses ongoing work towards a system for multivariate polynomials.

1.2 Automation in Lean

Previous work outlines a number of approaches to proof automation in Lean 4, each with different advantages and preferred applications. The Ring of Integers project [1] covers proof automation methods in Lean extensively and may be referenced for a more technical address of the topic.

This blueprint will introduce and evaluate proof automation strategies in previous work as it relates to our approach. We broadly refer to these strategies as *proof by reflection*.

One proof-by-reflection paradigm is white-box automation [2], in which the automation procedure is transparent and simple enough for users to trace and predict how an application will deal with their goals. Although these solutions tend to be narrow-application in terms of the goals they can deal with, they are highly extensible to different contexts since the user can understand and customize them. Search tactics such as `aesop` [2] and reflection models such as LeanSSR [3] are common examples of such automation. While alternative proof-by-reflection paradigms such as LLMs are more general-application, they tend to be more brittle to changes in context and difficult to debug due to their complexity. There have even been recent efforts to make these solutions more usable by integrating them as assistants in the interactive-proving process, using them to resolve subgoals instead of relying on them for complex theorems. Lean Copilot [4], for instance, integrates their LLM implementation with aesop's search, resulting in a white-box solution with some of the general-application benefits of LLM automation.

Popular implementations also tend to be native to Lean since they are easier to use, entail less dependencies, and enjoy the benefits of Lean's powerful tactics and metaprogramming. For instance, Lean Copilot [4] implements their Lean declarations in C++, to which Lean compiles directly. Moreover, LeanSSR [3] emphasizes how Lean is a great candidate for implementation of their reflection model because of its powerful metaprogramming support. LeanSSR also discusses how Lean's flexible tactics for goal manipulation result in convenient "last-mile automation" strategies, where the user only needs to manipulate the goal to a point where it can be automatically resolved by some form of reduction. For example, the built-in tactics `rfl` and `simp` automate a variety of goals by reducing them according to simple rules. Our native approach makes generous use of these last-mile automation strategies. Overall, it seems that native, white-box automation is a leading contender for Lean.

Type-class inference, which our project employs to construct computable representations, seems unexplored as a search method compared to metaprogramming and search tactics. We find in our project that it greatly simplifies resolving goals that can be split into similar subgoals without much additional verification at each step. However, it requires additional help from tactics when verification beyond the scope of the procedure is necessary at any step (see Section 2.3).

1.3 Polynomials in Lean

Previous work with polynomials in Lean highlights how a lack of computable representations can limit the impact of a formalization. As described by Wieser [5], polynomials in Mathlib are non-computable because they are implemented using the `Finsupp` type for finitely supported functions, whose underlying support set may not have decidable membership. An early Lean 3 formalization [6] aimed to prove the Mason Stothers Theorem. The project achieved this goals, though it first had to build up the necessary background in number theory and polynomials. A later project [7] which achieved the same formalization using Lean 4's Mathlib noted that the previous project's rejection of established machinery limited its application and reusability. This is because its results cannot be easily related to the standard Mathlib definitions used in other projects.

Some projects take advantage of the computational benefits of machine formalization, implementing algorithms on polynomials in Lean. One project implemented Buchberger's Algorithm [8] but noted that Mathlib's non-computable polynomial representation made it ineffective for their application. The authors implemented their own polynomials for use in their implementation, limiting its reusability similarly to the projects dealing with the Mason Stothers Theorem.

Another project [9] implemented algorithms for finding solutions to univariate polynomials, but built their mathematical machinery—all the way from natural numbers—from scratch. Although an impressive look into what it takes to formalize such mathematical background, this definitional separation limits the project in the same ways as the previous one.

Recent work focuses on more computable representations and automations for polynomials in Lean. Davenport [10] outlines the design decisions towards implementing such a representation and some operations on it. However, he does not detail any proof of correctness for those operations, and instead discusses verification using SageMath. Another approach to automation is Wieser's `polyrith` tactic [11], which uses SageMath to compute necessary parameters for resolving a goal with Mathlib's `linear_combination` tactic. It is a highly effective white-box method for polynomials over fields. In both these methods, proof automation is non-native. Our approach introduces a computable representation and correctness-verified operations all native to Lean. Furthermore, the representations do not require polynomials defined over fields, only commutative semirings in the most specific cases, making the solution more general than polyrith in this aspect.

The Ring of Integers project [1] briefly discusses construction of computable representations based on expressions in the current goal as a proof automation method. They also discuss their list-based implementation of a computable representation for polynomials which is similar to our list-based implementation. However, they explain as a limitation that their implementation does not provide such a way to automatically construct this representation. This type of automated construction is central to our approach and is achieved using the unexplored method of type-class inference (recall Section 1.2). For these reasons, we consider this aspect the most novel contribution of our work.

It should be noted that the powerful `grind` tactic [12] for automated reasoning over finite algebras became available in Lean during work on this project. For its Gröbner basis computations, the system ships with a computable representation of polynomials implemented by the `Poly` type (see `Poly.lean`) and automated construction methods for it. It is still unclear how those automated construction methods compare to those used in our project, though future work may explore this question as well as how adoption of the convenient `Poly` type could improve our system.

2 Reflection by Inference

2.1 Representation Inference

Lean's type-class inference is a powerful tool for automatically constructing instances of a parameterized type. Our approach to proof-by-reflection, which we will call *reflection by inference*, employs type-class inference to construct computable representations of target properties. First, we will briefly discuss type-class inference. Instance declarations are the blueprint for this process; each declaration states the parameter of the type it instantiates, admits any type class instances it requires as input, and specifies how to construct the target instance from these inputs. As displayed in Figure 2, type-class inference is a depth-first search which, at each step, checks all instance declarations that match the current goal. For more details on type-class inference, consult the *Theorem Proving in Lean 4* handbook [13].

Lean's type-class inference is central to our reflection model and we will refer to it as the *inference procedure*. In our case, the types we will construct are of computable representations of the target properties parameterized over polynomials. We will refer to these computable represention as *representations* and the type classes that contain instances of them as *reflection classes*. As addressed in Figure 1, reflection classes are generic to the representations they

Figure 1: Visualization of the reflection-by-inference model.

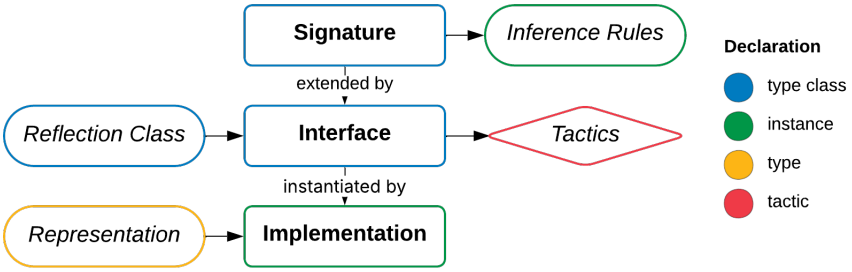
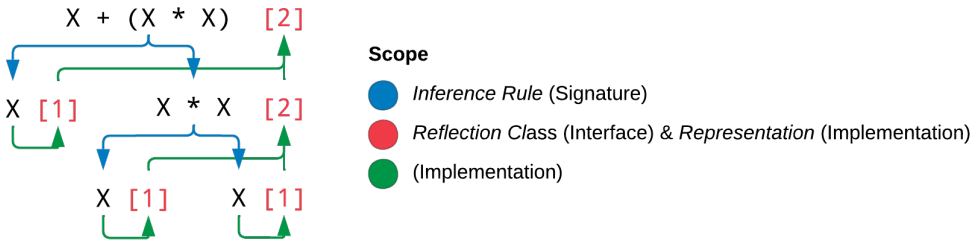


Figure 2: The inference procedure constructs an upper bound on the degree of a polynomial.



contain. It follows that the inference procedure is generic to computable representations of the target properties. We will refer to instance declarations for reflection classes as *inference rules*. Once the reflection classes and inference rules are declared, we may implement *tactics* that perform the inference procedure to resolve goals. These tactics rewrite the target property as the constructed computable representation so that the goal can be resolved by last-mile automation tactics. Recall that the LeanSSR project [3] discusses last-mile automation.

2.2 Reflection Model

The reflection-by-inference model in Figure 1 guides our implementation of the system for polynomials. We ensure levels of modularity with respect to the target property and the representation of that property by specifying three levels of abstraction: The *signature* declares inference rules for generic reflection classes. The *interface* extends multiple signatures with a specified reflection class that asserts the target property. At this level, tactics may be implemented for generic representations. The *implementation* instantiates an interface with a specified representation of the target property and implements the rules declared in the signatures.

2.3 Sensitive Inference Procedure

Although the inference procedure is effective for target properties with inference rules that require only the simple input instances without extra conditions, it is limited for properties that require extra assumptions at each step. For example, inference for exact degrees and leading coefficients

requires that leading terms do not cancel at each `+`, `*`, or `^` step. To address this issue, we implement the *sensitive inference procedure* which is invoked by Tactic 2.1.

Tactic 2.1 (`infer_instance_trying`)**.** Performs a depth-first search which, at each step, applies a provided helper tactic `t` then checks all instance declarations that match the current goal. Unlike Lean's type-class inference, this procedure supports instance declarations that require assumptions that cannot be resolved by type-class instances, handling them with the helper tactic instead. The syntax is `infer_instance_trying <:> t`.

```
open Nat

class IsDouble (n :  ) where
  m :
  h : n = 2 * m

instance (h : n % 2 = 0) : IsDouble n :=
   n / 2, (Nat.mul_div_cancel' (dvd_of_mod_eq_zero h)). symm

example : IsDouble 4 := by infer_instance_trying <:> decide
```

3 Univariate System

3.1 Inference Rules

TODO

3.2 Reflection Classes

TODO

3.3 Representations

TODO

3.4 Tactics

3.4.1 Utility tactics. Utility tactics perform useful rewriting and last-mile simplification steps. They appear frequently in those reflection tactics that perform the inference procedure.

Tactic 3.1 (`poly_rfl_rw`)**.** Performs trivial rewrites to transform polynomials into the form expected by the inference procedure. For instance, `0` is rewritten as `C 0`.

Tactic 3.2 (`poly_rfl_dsimp`)**.** Simplifies the expression resulting from the inference procedure so that the computable representation is visible. This consists primarily of unfolding reflection class instances into the values they contain.

Tactic 3.3 (`poly_rfl_with`)**.** Frames a reflection tactic `t` with rewriting and last-mile simplification steps from Tactic 3.1 and Tactic 3.2. The syntax is `poly_rfl_with <:> t`.

Tactic 3.4 (`poly_infer_try`)**.** For use as a helper tactic in the sensitive inference procedure (see Tactic 2.1). Invokes Tactic 3.2 to unfold the expression resulting from the previous step of the sensitive inference procedure, preparing the goal for the next step.

3.4.2 Reflection tactics. Reflection tactics perform the inference procedure to resolve target goals by transforming them so that they can be resolved with last-mile automation tactics such as `trivial`, `simp`, and `norm_num`. The code samples are adapted from `Demo/Polynomial.lean` and rely on the following preamble.

```
import AutomatePolynomial.Reflection.Polynomial.Basic
open Polynomial Rfl
```

Tactic 3.5 (`poly_rfl_degree_le`)**.** Resolves goals of the form $\texttt{degree}(p) \leq n$ for univariate polynomials p and some n which is either a natural number or the bottom member $\bot$.

```
section DegreeLe
variable [Semiring R]
example : (0     : R[X]).degree    := by poly_rfl_degree_le; trivial
example : (1     : R[X]).degree  0 := by poly_rfl_degree_le; trivial
example : (X     : R[X]).degree  1 := by poly_rfl_degree_le; trivial
example : (X ^ 2 : R[X]).degree  2 := by poly_rfl_degree_le; trivial
example : (X + 1 : R[X]).degree  1 := by poly_rfl_degree_le; trivial
end DegreeLe
```

Since the following tactics deal with properties that depend on whether leading terms cancel—degrees and leading coefficients—they will rely on the sensitive inference procedure (see Section 2.3) in cases where the polynomial contains recursive cases such as `+`, `*`, or `^` (disregarding `X ^ n` which is handled separately). Tactics with the suffix `_trying` perform the sensitive inference procedure to handle these cases. Alternatively, the sensitive inference procedure may be avoided by instead infering the coefficients of the polynomial and then deriving the target property from that. Tactics with the suffix `_of_coeffs` use this alternative when provided the the target representation of the coefficients, such as `CoeffsList`.

Tactic 3.6 (`poly_rfl_degree_eq`)**.** Resolves goals of the form $\texttt{degree}(p) = n$ for univariate polynomials p and some n which is either a natural number or the bottom member $\bot$. In any case where we want $\texttt{degree}(p) \neq \bot$, we must admit that the the semiring is nontrivial, as in section `DegreeEqNontrivial`.

```
section DegreeEq
variable [Semiring R]
example : (0 : R[X]).degree =   := by poly_rfl_degree_eq
end DegreeEq

section DegreeEqNontrivial
variable [Semiring R] [Nontrivial R]
example : (1     : R[X]).degree = 0 := by poly_rfl_degree_eq
example : (X     : R[X]).degree = 1 := by poly_rfl_degree_eq
example : (X ^ 2 : R[X]).degree = 2 := by poly_rfl_degree_eq
example : (X + 1 : R[X]).degree = 1 := by poly_rfl_degree_eq_trying <:> poly_infer_try
end DegreeEqNontrivial

section DegreeEqOfCoeffs
example : (X + 1 :  [X]).degree = 1 := by poly_rfl_degree_eq_of_coeffs VIA CoeffsList;
    simp; trivial
end DegreeEqOfCoeffs
```

Tactic 3.7 (`poly_rfl_leading_coeff`)**.** Resolves goals of the form `leadingCoefficient`$(p) = c$ for univariate polynomials p and members of the relevant semiring c. When the definition of p contains recursive cases such as `+`, `*`, or `^` (disregarding `X ^ n` which is handled separately), we must admit that the semiring is nontrivial, as in section `LeadingCoeffNontrivial`.

```
section LeadingCoeff
variable [Semiring R]
example : (0     : R[X]).leadingCoeff = 0 := by poly_rfl_leading_coeff
example : (1     : R[X]).leadingCoeff = 1 := by poly_rfl_leading_coeff
example : (X     : R[X]).leadingCoeff = 1 := by poly_rfl_leading_coeff
example : (X ^ 2 : R[X]).leadingCoeff = 1 := by poly_rfl_leading_coeff
end LeadingCoeff

section LeadingCoeffNontrivial
variable [Semiring R] [Nontrivial R]
example : (X + 1 : R[X]).leadingCoeff = 1 := by poly_rfl_leading_coeff_trying <:>
    poly_infer_try
end LeadingCoeffNontrivial

section LeadingCoeffEqOfCoeffs
example : (X + 1 :  [X]).leadingCoeff = 1 := by poly_rfl_leading_coeff_of_coeffs VIA
    CoeffsList; simp
end LeadingCoeffEqOfCoeffs
```

The remaining tactics are generic to the representation of the target property and require the user to provide the target representation, such as `CoeffsList` or `EvalArrow`.

Tactic 3.8 (`poly_rfl_coeff`)**.** Resolves goals of the form `nthCoefficient`$_n(p) = c$ for univariate polynomials p, natural numbers n, and members of the relevant semiring c. Note that we admit that the semiring is commutative.

```
section Coeffs
variable [CommSemiring R]
example : (0     : R[X]).coeff 1 = 0 := by poly_rfl_coeff VIA CoeffsList
example : (1     : R[X]).coeff 1 = 0 := by poly_rfl_coeff VIA CoeffsList; trivial
example : (X     : R[X]).coeff 1 = 1 := by poly_rfl_coeff VIA CoeffsList; trivial
example : (X ^ 2 : R[X]).coeff 1 = 0 := by poly_rfl_coeff VIA CoeffsList; trivial
example : (X + 1 : R[X]).coeff 1 = 1 := by poly_rfl_coeff VIA CoeffsList; simp
end Coeffs
```

Tactic 3.9 (`poly_rfl_eval`)**.** Resolves goals of the form $p(c) = c'$ for univariate polynomials p and members of the relevant semiring c and c'. Note that we admit that the semiring is commutative.

```
section Eval
variable [CommSemiring R]
example : (0     : R[X]).eval 1 = 0 := by poly_rfl_eval VIA EvalArrow
example : (1     : R[X]).eval 1 = 1 := by poly_rfl_eval VIA EvalArrow
example : (X     : R[X]).eval 1 = 1 := by poly_rfl_eval VIA EvalArrow
example : (X ^ 2 : R[X]).eval 1 = 1 := by poly_rfl_eval VIA EvalArrow; simp
example : (X + 1 : R[X]).eval 1 = 2 := by poly_rfl_eval VIA EvalArrow; norm_num
end Eval
```

Tactic 3.10 (`poly_rfl_expand`)**.** Resolves goals of the form $p = q$ for a univariate polynomial p and an equivalent polynomial q in expanded form. Note that we admit that the semiring is commutative.

```
section Expand
variable [CommSemiring R]
example : (C 2 + X : R[X]) = X + C 2 := by poly_rfl_expand VIA CoeffsList; simp;
    poly_unfold_expand; simp
example : (X * C 2 : R[X]) = C 2 * X := by poly_rfl_expand VIA CoeffsList; simp;
    poly_unfold_expand; simp
example : (X + X   : R[X]) = C 2 * X := by poly_rfl_expand VIA CoeffsList; simp;
    poly_unfold_expand; norm_num
end Expand
```

4 Multivariate System

Current work focuses on generalizing the capabilities of the system for univariate polynomials to a system for multivariate polynomials. Progress on this system is available in the `feature/mvpoly` branch.

References

[1] Anne Baanen, Alain Chavarri Villarello, and Sander R. Dahmen. Certifying rings of integers in number fields. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2025, page 50–66, New York, NY, USA, 2025. Association for Computing Machinery.

[2] Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, page 253–266, New York, NY, USA, 2023. Association for Computing Machinery.

[3] Vladimir Gladshtein, George Pîrlea, and Ilya Sergey. Small scale reflection for the working lean user, 2024.

[4] Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Lean copilot: Large language models as copilots for theorem proving in lean. NeuS 2025, 2025.

[5] Eric Wieser. Computation models for polynomials and finitely supported functions. https://github.com/leanprover-community/mathlib4/wiki/Computation-models-for-polynomials-and-finitely-supported-functions, 2023.

[6] Jens Wagemaker. A formally verified proof of the mason-stothers theorem in lean, 2018.

[7] Jineon Baek and Seewoo Lee. Formalizing mason-stothers theorem and its corollaries in lean 4, 2024.

[8] Markos Dermitzakis. An implementation of buchberger's algorithm in lean, 2019.

[9] Nicholas Dyson, Benedikt Ahrens, and Jacopo Emmenegger. The solutions to single-variable polynomials, implemented and verified in lean, 2022.

[10] James Harold Davenport. First steps towards computational polynomials in lean, 2024.

[11] Dhruv Bhatia, Eric Wieser, Mario Carneiro, and Thomas Zhu. polyrith tactic. https://leanprover-community.github.io/mathlib4_docs/Mathlib/Tactic/Polyrith.html, 2022.

[12] *The Lean Language Reference*, chapter The grind tactic. GitHub.

[13] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. *Theorem Proving in Lean 4*, chapter Type Classes. GitHub.