Reflection in Lean 4: Automated Reasoning for Polynomials

Liam Schilling, advised by Quang Dao

School of Computer Science, Carnegie Mellon University

Introduction

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.

Representation Inference

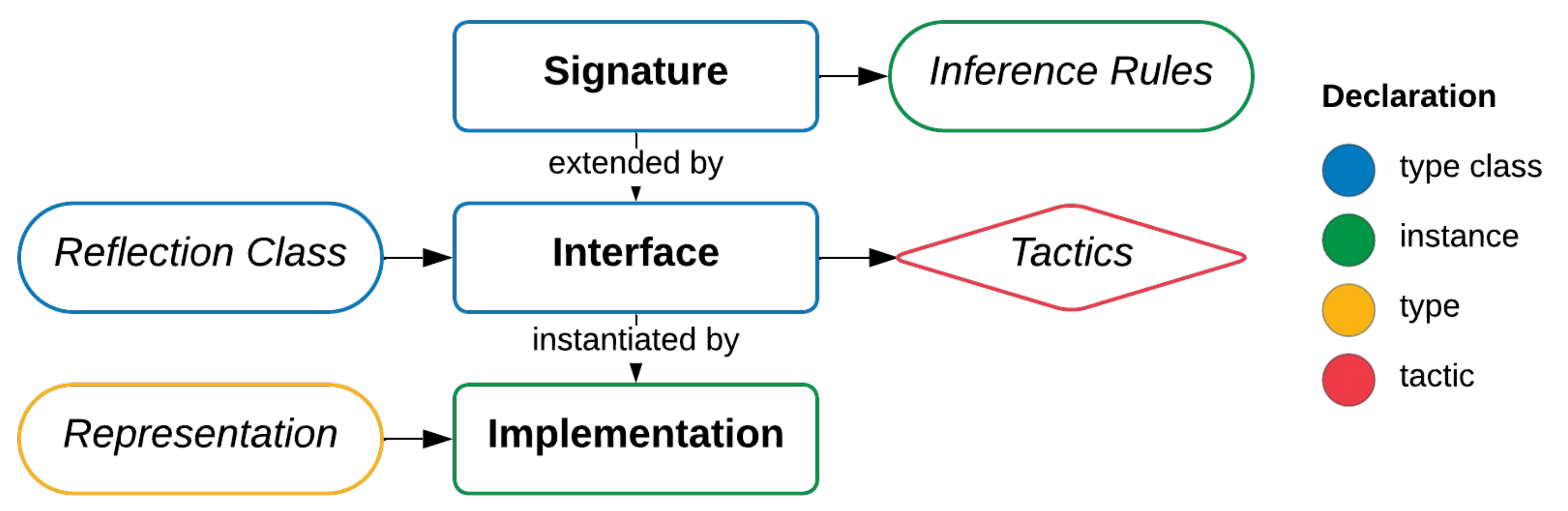
We employed Lean's type-class inference to trace an expression of type "polynomial" along base (e.g. constants, variables) and recursive (e.g. addition, multiplication) cases according to corresponding type-class instances. These instances then construct and verify the representation from the bottom up.

To handle representations requiring more strict verification–degree equality for example, which requires that leading terms do not cancel–we implemented the tactic `infer_instance_trying` which tries a helper tactic on any subgoal where Lean's inference fails.

Reflection Model

We specify three levels of abstraction.

- A **Signature** declares necessary *Inference Rules*, which yield instances for a generic type class.
- The **Interface** extends multiple signatures with a specified *Reflection Class* asserting the target property. At this level, *Tactics* to automate proof of goals can 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.

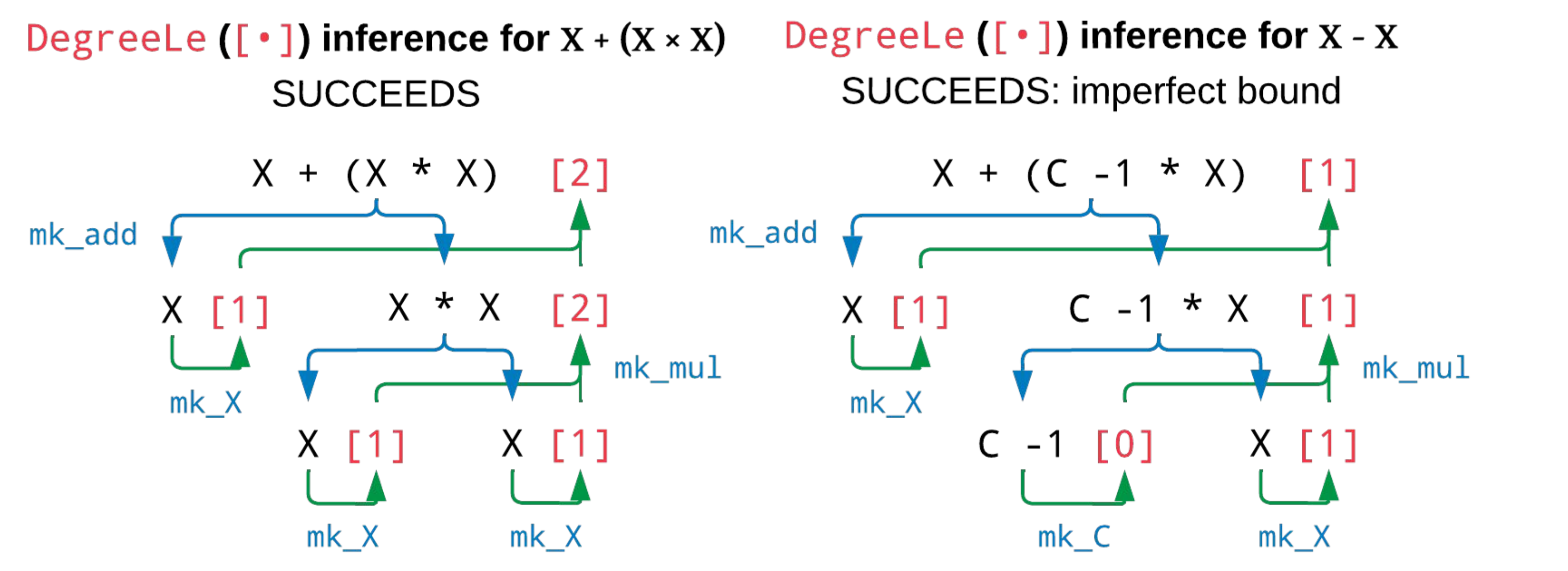


Reflection Systems

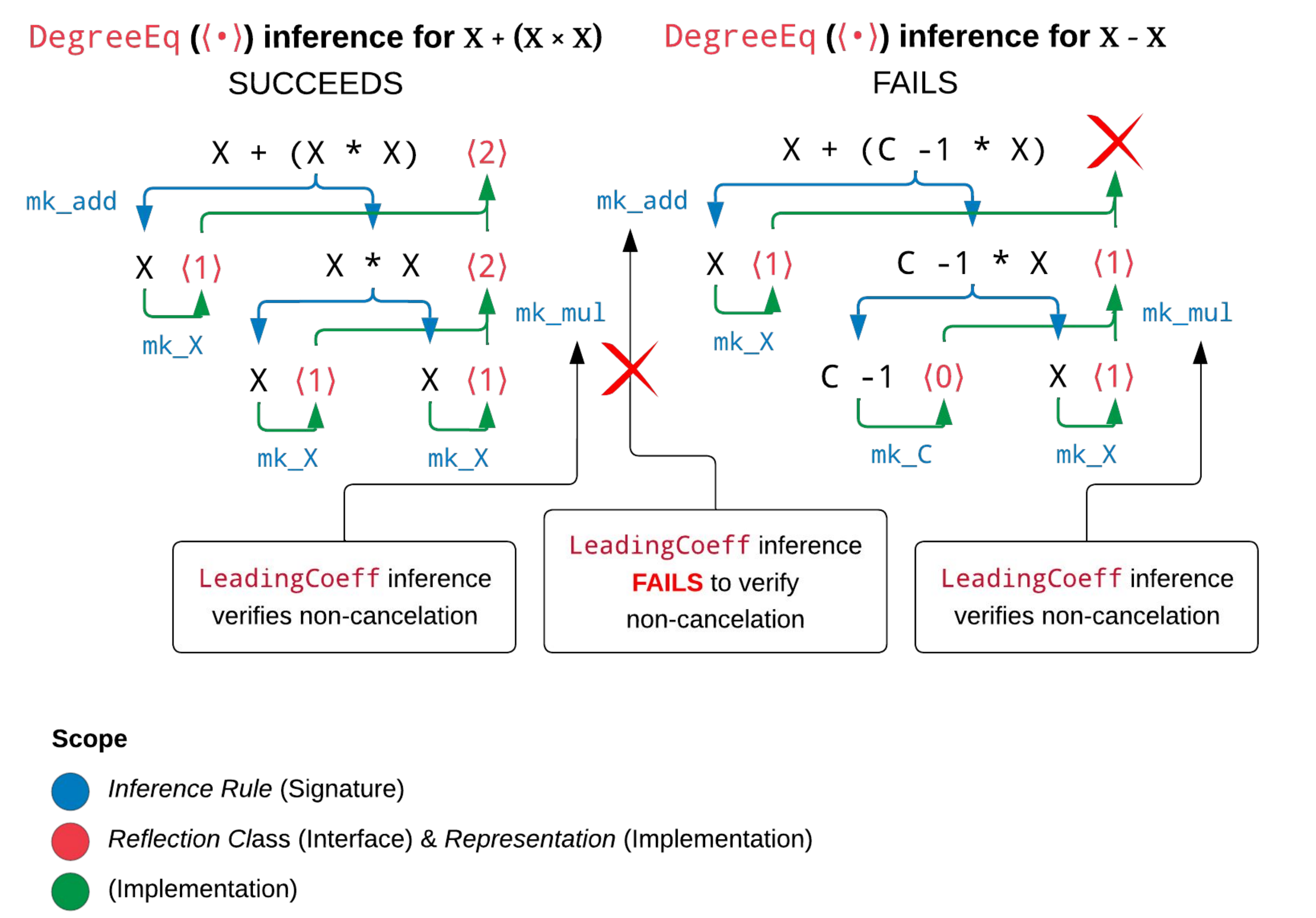
We identified five properties relevant to univariate and multivariate polynomials (listed reflection classes are for univariate polynomials).

- `DegreeEq`: exact degree (*Sensitive*)
- `DegreeLe`: greedy upper bound on degree
- `LeadingCoeff`: exact leading coefficients (*Sensitive*)
- `Coeffs`: exact specification of all coefficients
- `Eval`: exact specification of evaluations at all points

Non-*Sensitive* inference rules do not depend on leading term cancellations. This enables quick verification of an upper bound with `DegreeLe` that is only imperfect when leading terms cancel.

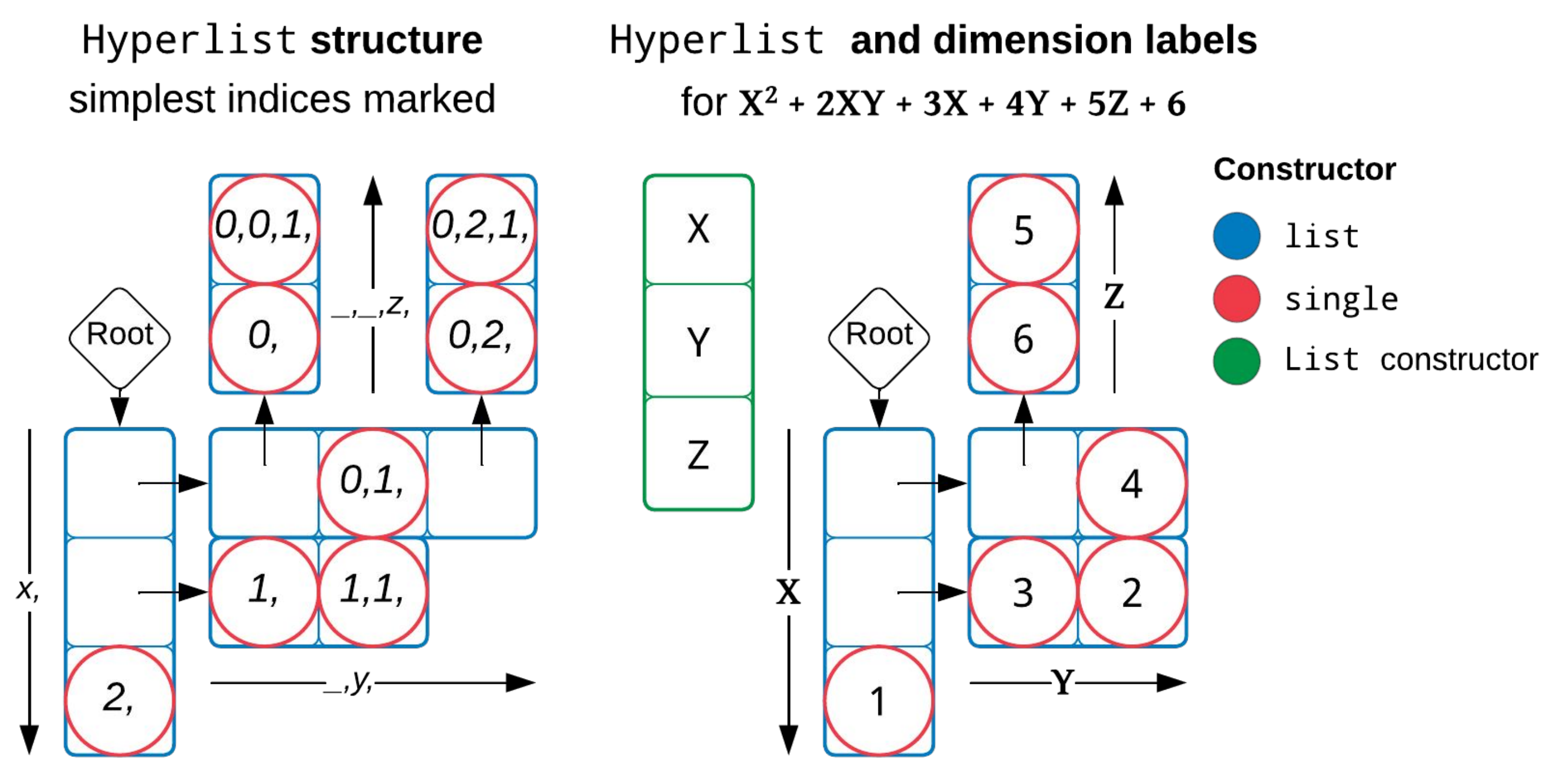


Sensitive inference rules require proof that leading terms do not cancel. `infer_instance_trying` treats these problems, when represented as equivalent propositions on `LeadingCoeff` instances, as typical subgoals and verifies them when they hold.


DegreeEq (⟨•⟩) inference for X + (X × X)
SUCCEEDS
X + (X * X) ⟨2⟩
mk_add
X ⟨1⟩
X * X ⟨2⟩
mk_X
mk_mul
X ⟨1⟩
X ⟨1⟩
mk_X
mk_X
DegreeEq (⟨•⟩) inference for X - X
FAILS
X + (C -1 * X)
mk_add
X ⟨1⟩
C -1 * X ⟨1⟩
mk_X
mk_mul
C -1 ⟨0⟩
X ⟨1⟩
mk_C
mk_X
LeadingCoeff inference verifies non-cancelation
LeadingCoeff inference FAILS to verify non-cancelation
LeadingCoeff inference verifies non-cancelation
Scope
Inference Rule (Signature)
Reflection Class (Interface) & Representation (Implementation)
(Implementation)

Representations

We represented evaluations as lambdas and univariate coefficients as lists. We implemented unboundedly-higher-dimensional lists for multivariate coefficients, in which index `[i,j,...k]` holds the `X^i*Y^j*...Z^k` coefficient. Such a `Hyperlist` is a tree with constructors `list`, a branching list of subhyperlists, and `single`, a leaf holding a single value. A separate `List` labels its dimensions, the variables of the polynomial it represents.



Results

The resulting systems automate proof of degrees, coefficients, evaluations, and expansions for univariate and multivariate polynomials in various contexts. The following *tactics* automate proof of the **goals** in the context of a nontrivial semiring.

```
example : ((X + 1) ^ 2 * (X ^ 2 + X)).degree = 4 :=
  by poly_rfl_degree_eq_trying <:> poly_infer_try; trivial

example : ((X + 1) ^ 2 * (X ^ 2 + X)).coeff 3 = 1 + 1 + 1 :=
   by poly_rfl_coeff VIA CoeffsList; simp

example : (X + 1) ^ 2 * (X ^ 2 + X) =
     X ^ 4 + (1 + 1 + 1) * X ^ 3 + (1 + 1 + 1) * X ^ 2 + X :=
   by poly_rfl_expand VIA CoeffsList
   simp; poly_unfold_expand; simp[add_assoc]
```

Discussion

Future work will improve the efficiency and strength of the systems in this project. Sparse and array representations of coefficients are priorities. Search tactics beyond `infer_instance_trying` will also be explored for more complete automation.

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.