# How to design tools for formalization

Robert Y. Lewis

BROWN

ICERM
May 13, 2026

# The plan

I want to:

- Introduce what tactics actually are.
- Describe some styles of tactics.
- Suggest guidelines for designing new tactics.

Motivation: this workshop is about tools and techniques for analysis. To think up useful and feasible tools, we should understand a bit about how those tools come into existence!

# What are tactics?

# What are metaprograms?

# Lean syntax and expressions

The syntax that you're used to writing in a Lean file is superficial.

Elaboration is the art of going from input syntax to abstract syntax.

This happens in all languages to some degree, but especially complicated in languages with lots of implicit information.

# The Expr type

Lean describes its own abstract syntax in the type `Lean.Expr`.

An expression is either

- a named constant (with universe parameters filled in), or
- a bound variable, or
- a lambda expression (with variable type and body, both expressions), or
- the application of one expression to another, or . . .

Type-correct expressions are a subset of expressions. Note that type correctness depends on an environment.

# Declarations

A declaration is (essentially) a triple of a name, an expression (the type), and an expression (the body).

```
theorem theorem_name : TheoremType := theorem_body
```

`theorem`, `def`, etc. take surface syntax, elaborate it, type check it, build a declaration, and add that to the environment. (This is a metaprogram!)

# Tactic scripts

```
theorem theorem_name : TheoremType := by
  intro x
  apply theorem_body
```

A tactic script must meet this same interface. It is essentially a program that outputs an expression.
Slightly more technically:

- Lean creates a metavariable `?g : TheoremType`
- Each tactic accesses the current goals as a list of metavariables, and assigns them to expressions (possibly containing new metavariables)
- When no goals remain, `?g` is the body of `theorem_name`

# Proof by term assembly

# Term assembly

Many tactics take this more or less literally: look at the proof state, (partially) assemble a term.

- “atomic” tactics: `intro`, `refine`, `apply`, . . .
- “library-aware” tactics: `norm_cast`, `gcongr`, . . .
- “recursive” tactics: `interval_cases`, . . .

Especially common in nonterminal tactics and tactics that don’t do much *searching*.

# Tactic syntax

Many mathlib tactics make only small modifications to the proof term. They're mostly there for the convenience of their surface syntax.

This is very important!

# Meta vs object

Lean 4 doesn't have a hard boundary between "object" and "meta" code.

Still: what would you need to do to prove correctness of a tactic like `interval_cases`?

- Semantics for expressions, proof states, environments, etc
- Proof APIs for lots of meta functions
- . . .

What does the spec even look like?

# Proof by reflection

# When proof meets metaprogramming

What if we restrict the language of interest down from “all Lean expressions” to “formulas in some fixed domain”?

For example, equalities on ring expressions.

- `RingSyntax : Type`
- `interp {α : Type} [Ring α] : RingSyntax → α`
- `normalize : RingSyntax → RingSyntax`
- `normalize_correct (r1 r2 : RingSyntax) : normalize r1 = normalize r2 → interp r1 = interp r2`

# Reflection

Faced with a goal

```
α : Type
inst : Ring α
x y z : α
⊢ x * (y + z) = z * x + 2 * y * z - z * y
```

Change it to the definitionally equal

```
α : Type
inst : Ring α
x y z : α
⊢ interp ⟦x * (y + z)⟧ = interp ⟦z * x + 2 * y * z - z * y⟧
```

where $[\![\cdot]\!]$ is the “inverse” of `interp`. This we can compute at the meta level!

# Computation

```
α : Type
inst : Ring α
x y z : α
⊢ interp ⟦x * (y + z)⟧ = interp ⟦z * x + 2 * y * z - z * y⟧
```

Applying `normalize_correct`, this goal becomes

```
α : Type
inst : Ring α
x y z : α
⊢ normalize ⟦x * (y + z)⟧ = normalize ⟦z * x + 2 * y * z - z * y⟧
```

which is an equality over `RingSyntax` that we can prove by `rfl`.

# Tradeoffs

- Proof terms are very small
- Proof effort is entirely at the object level, which is more pleasant
- Immune to certain kinds of failure
- Relies on kernel computation

# Proof by certificate

# Search tactics

We’ve mostly been thinking about tactics whose proof path is “linear.”

# Search tactics

We've mostly been thinking about tactics whose proof path is "linear."

How would you implement a tactic like `linarith`?

# Search tactics

We've mostly been thinking about tactics whose proof path is "linear."

How would you implement a tactic like `linarith`?

- Reflection: implement syntax of linear expressions, define decision procedure, prove search is complete, run search in kernel
- Prove-as-we-go: build up lots of maybe-unnecessary "state" in the proof context?

# Certificates

Facing goal
$h_1 : a_1^1 x_1 + a_2^1 x_2 + \ldots + a_n^1 x_n \bowtie^1 0$
$h_2 : a_1^2 x_1 + a_2^2 x_2 + \ldots + a_n^2 x_n \bowtie^2 0$
$\ldots$
$h_k : a_1^k x_1 + a_2^k x_2 + \ldots + a_n^k x_n \bowtie^k 0$
$\vdash$ False
with $\bowtie^i \in \{<, \leq, =\}$, suppose we find

- $c_1, \ldots, c_k \geq 0$ such that
- $\sum_{i=1}^{k} c_i (a_1^i x_1 + a_2^i x_2 + \ldots + a_n^i x_n) = 0$
- and for at least one $i$, $c_i > 0$ and $\bowtie^i$ is $<$.

Given this list of coefficients, I can easily produce a proof term of `False`.

# Oracular search

Insight: it doesn't matter where these coefficients come from.

- Mathlib has two such oracles, you could prove things about them if you want
- Computer algebra tools can do it
- You could check the certificate by reflection, or by creating a proof term

Separating search from certification.

# So you want to design a tactic?

# Guidelines

Metaprograms can do almost anything, but they can't do more than you could (in principle) do by hand.

Have an idea for a tactic? Write out examples, with and without completed proofs.

# Scoping

It’s better for a tactic to do one thing perfectly than three things reasonably well.
Predictability for users is key.

Accidental successes on problems that are out of scope are confusing.

# Design vs implementation

Try not to commit to an implementation up front.

Eventually the implementation will determine what's easy or hard to do. But describing the proper front-end interface and scope is difficult on its own.

Do consider if, how, and when your tactic *searches*.

# Library integration

Metaprograms can be library-aware and library-integrated. They’re first class citizens in mathlib.

Don’t hesitate to dream about domain-specific tools.

# Additional resources

# Additional resources

- Metaprogramming in Lean 4
- Tactic Programming Guide
- Hitchhiker's Guide