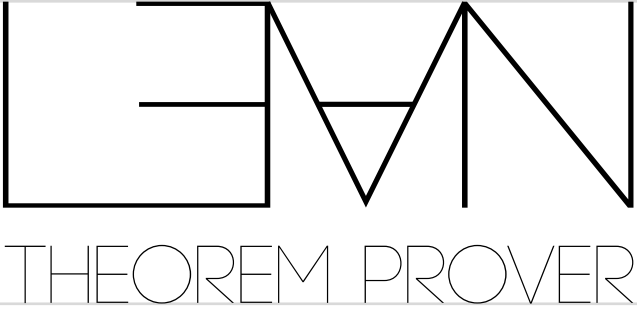
Karlsruhe Institute of Technology

Theorembeweiserpraktikum

Metaprogramming in Lean

Jakob von Raumer, Sebastian Ullrich | SS 2022



Aesop News

- All our extensions but `elimAny` upstreamed and enabled by default
- Probability for `unsafe` now defaults to `50%`

Tactics, How Do They Work?

- Tactics modify our proof goals and contexts to incrementally solve a problem.
- How is the final term proof assembled?
- Metavariables: Holes in term expressions which need to be filled.
- Example:

```
example {a b c : α} (h : a = b) (h' : b = c) : a = c := by ...
```

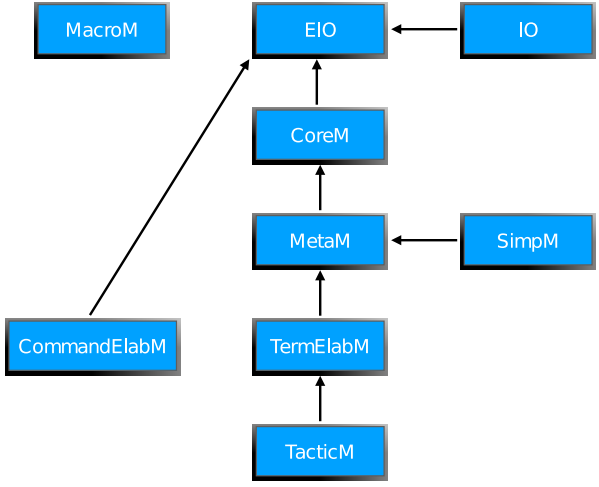
  creates a hole `?m1 : a = c`. Using `apply Eq.trans` will *fill* this hole with `?m1 := Eq.trans ?m2 ?m3` while creating new holes `?m2 : a = ?m4` and `?m3 : ?m4 = c` and `?m4 : α`.
- Tactics return unsolved metavariables as goals, so after the last goal is closed, no unsolved metavariables remain.

What are Monads?

- Lean is a purely functional language (deterministic, no side-effects)!
- How to circumvent this in cases where we really need a state, like when writing tactics? $\rightsquigarrow$ Replace functions $\alpha \to \beta$ by functions $\text{State} \times \alpha \to \text{State} \times \beta$
- *Monads* are the abstract (and more general) concept behind this.
- Lean implements `do` notation to work "inside" monads:

```
example (a : StateM Nat α) : StateM Nat α := do
  let a' ← a
  StateT.set 5
  return a'
```

Lean's Monad Stack



Writing our own Introduction Tactic

- Given a hole `?m1 : (a : α) → β a` we want to
  1. solve this hole with `?m1 := fun (a : α) => ?m2` and
  2. return `?m2` as a new goal, where `a : α` is in the local context.

Writing our own Introduction Tactic

```
syntax (name := myIntro) "myIntro" : tactic

@[tactic myIntro] def elabMyIntro : Tactic -- `Tactic` is a shortcut for `Syntax → TacticM Unit`
  | `(tactic|myIntro) => do  -- We match the incoming syntax
    let mVarIds ← getUnsolvedGoals -- We get a List of unsolved goals
    match mVarIds with
    | [] => throwNoGoalsToBeSolved -- If there are goals to be solved we throw an error
    | mVarId :: otherGoals => -- Now we have the metavariable that we want to work on and "other goals"
      match ← getMVarType mVarId with -- Get the type of the metavariable
      | Expr.forallE n h b d => -- Match the type on a `∀ ...`, otherwise throw an error
        withLocalDecl n d.binderInfo h fun ld => do -- For the new metavar, we add a new declaration
          let bodyMVar ← mkFreshExprSyntheticOpaqueMVar (b.instantiate1 ld) -- Create the new metavar, instantiate `ld`
          let val ← mkLambdaFVars #[ld] bodyMVar -- Build the lambda term which we want to assign to the original metavar
          assignExprMVar mVarId val
          setGoals (bodyMVar.mvarId! :: otherGoals)
      | _ => throwTacticEx `myIntro mVarId "tactic not applicable"
  | _ => throwUnsupportedSyntax
```