Toward AI for Lean, via metaprogramming

Robert Y. Lewis

Vrije Universiteit Amsterdam

March 28, 2018
Introduction

What this talk is not about:

- novel AI techniques
- novel AI applications
- finished work
Introduction

What this talk is not about:

- novel AI techniques
- novel AI applications
- finished work

What this talk is about:

- “easy” AI applications in ITP
- stress-testing Lean’s tactic framework
- components of something larger
Table of Contents

Introduction

Lean and metaprogramming

An internal relevance filter

Connecting Lean and Mathematica

An external relevance filter
Background: Lean

Lean is a new interactive theorem prover, developed principally by Leonardo de Moura at Microsoft Research, Redmond.

Calculus of inductive constructions with:

- non-cumulative hierarchy of universes
- impredicative Prop
- quotient types and propositional extensionality
- axiom of choice available

See http://leanprover.github.io
Expression evaluation

Lean pre-expression

1 + 1

elaborator

Lean expression

add nat nat.has_add (one nat nat.has_one) (...)

kernel

Trusted type-checked expression

Trusted reduced expression

nat.succ (nat.succ nat.zero)

compiler

Compiled VM code

Untrusted reduced expression

2

VM
The Lean VM

- The VM can evaluate anything in the Lean library, as long as it is not noncomputable.
- It substitutes native nats, ints, arrays.
- It has a profiler and debugger.
- The VM is ideal for non-trusted execution of code.
Definitions tagged with `meta` are “VM only,” and allow unchecked recursive calls.

```lean
meta def f : ℕ → ℕ
| n := if n=1 then 1
  else if n\%2=0 then f (n/2)
  else f (3*n + 1)

#eval (list.iota 1000).map f
```
Metaprogramming in Lean

Question: How can one go about writing tactics and automation?

Lean’s answer: go meta, and use Lean itself.

Advantages:

▶ Users don’t have to learn a new programming language.
▶ The entire library is available.
▶ Users can use the same infrastructure (debugger, profiler, etc.).
▶ Users develop metaprograms in the same interactive environment.
▶ Theories and supporting automation can be developed side-by-side.
The strategy: expose internal data structures as meta declarations, and insert these internal structures during evaluation.

```lean
meta constant expr : Type
meta constant environment : Type
meta constant tactic_state : Type
meta constant to_expr : expr → tactic expr
```
Tactic proofs

```lean
meta def p_not_p : list expr → list expr → tactic unit
| []        Hs := failed
| (H1 :: Rs) Hs :=
do t ← infer_type H1,
    (do a ← match_not t,
         H2 ← find_same_type a Hs,
         tgt ← target,
         pr ← mk_app 'absurd [tgt, H2, H1],
         exact pr)
<|> p_not_p Rs Hs
```

```lean
meta def contradiction : tactic unit :=
do ctx ← local_context,
    p_not_p ctx ctx
```

```lean
lemma simple (p q : Prop) (h₁ : p) (h₂ : ¬p) : q :=
    by contradiction
```
Applications of metaprogramming

How far can we push this framework?

- simplification and normalization
- decision procedures
- connections to external software
- superposition prover

Target: a sledgehammer for Lean, without touching the Lean source code.
Relevance filtering

Given $P : \text{Prop}$, produce a list of declarations likely to be used to prove $P$.

We need to:

- map over the environment
- build a database of declarations
- define a relevance function
- find the top $k$ matches

```lean
meta def k_relevant_facts (target : expr) (k : ℕ) :
tactic (list name) := ...
```
Relevance filtering

Lean provides:

- **meta constant** `get_env : tactic environment`
- **meta constant** `environment.fold : environment → α →
  (declaration → α → α) → α`
- **meta constant** `rb_map : Type → Type → Type`
- **meta def** `array : Type → ℕ → Type`
- a VM implementation of arrays with destructive updates

For convenience, we add:

- **meta constant** `float : Type` and associated operations
Relevance filtering

Following Czajka and Kaliszyk (2018) we implement $k$ nearest neighbors and sparse naive Bayes classifiers.

```lean
meta def feature_distance (f1 f2 : name_set) : float :=
let common := f1.inter f2 in
(common.map compute_feature_weight).sum

meta def nearest_k (features : name_set) ⋯
{n} (names : array n name) (k : ℕ) :
list (name × float) :=
let arr_prs : array n (name × float) := ⟨λ i, ⋯⟩,
sorted := partial_sort
(λ n1 n2 : name × float, float.lt n2.2 n1.2) arr_prs k in
if h : k ≤ n then (sorted.take k h).to_list else
sorted.to_list
```
Relevance filtering

Downsides:

▶ inefficient (4-5 sec to build data structures)
▶ underdeveloped libraries
▶ depends on “hacked” version of Lean

Upsides:

▶ portable
▶ integrates with Lean library
▶ could potentially verify parts of the process
Computer algebra systems offer many things that proof assistants lack:

- fast computation
- huge(r) libraries of functions
- ease of use
- visualization and display
Connecting Lean and Mathematica

We¹ provide:

▶ an extensible procedure to interpret Lean in Mathematica
▶ an extensible procedure to interpret Mathematica in Lean
▶ a link allowing Lean to evaluate arbitrary Mathematica commands, and receive the results
▶ tactics for certifying results of particular Mathematica computations
▶ a link allowing Mathematica to execute Lean tactics and receive the results

¹RL + Minchao Wu (Australia National University)
The idea: many declarations in Lean correspond *roughly* to declarations in Mathematica.

We can do an approximate translation back and forth and verify post hoc that the result is as expected.

Correspondences, translation rules, checking procedures are part of a mathematical theory.
Calling Mathematica from Lean

Lean expr

Lean expr in MM syntax

Lean expr in MM syntax

Lean expr in Lean syntax

MM expr

MM expr

verification tactics
A relevance filter via Mathematica

Mathematica has various tools for “black box” machine learning.

We can build the data structures in Lean and send them to Mathematica for processing.
Calling Lean from Mathematica

Lean expr

Lean expr in MM syntax

Lean expr in MM syntax

MM expr

display or compute

MM expr in Lean syntax

MM expr
We can add rules to (try to) translate the resulting proof.

```mathematica
DiagramOfFormula[
  ForAll[{P, Q},
    Implies[
      Or[P, Q],
      Not[And[Not[P], Not[Q]]]
    ]
  ]
]
```
Calling Lean from Mathematica

With this link, we can access either relevance filter from within Mathematica.

- State a theorem in Mathematica, translate to Lean, and get relevant facts.
- Idea: make CAS useful for proof exploration instead of just computational exploration.
Calling Lean from Mathematica

```
SelectLeanPremises[ForAllTyped[{x, y}, real, Implies[x < y, Exists[{z}, And[x < z, z < y]]]]]
ennreal.densely_ordered._match_1 : ∀ (r : R), 0 ≤ r → ∀ (p : R), 0 ≤ p → (∃ (a : R), r < a ∧ a < p) → (∃ (a : ennreal), a < ennreal.of_real p ∧ ennreal.of_real r < a)
exists_pos_of_rat : ∀ {r : R}, 0 < r → (∃ (q : Q), 0 < q ∧ of_rat q < r)
```
Thanks for listening!

Questions, then skiing.