Formal Methods in Analysis: Plans and Prospects

Jeremy Avigad

Department of Philosophy and
Department of Mathematical Sciences
Carnegie Mellon University

June 23, 2018

The Plan

- Tell you about the grant that supported this workshop.
- Tell you about Lean.
- Discuss plans for both.

Formal Methods in Analysis

This workshop is associated with a grant from the Air Force Office of Scientific Research.

- program director: Fred Leve
- program: Dynamics and Control
- goals: dynamical systems, control theory, logic, constructive mathematics, formal verification
- two years, possible two-year renewal

Formal Methods in Analysis

Principal investigators and areas:

- Steve: HoTT, cohesion, and logics for dynamical systems
- André Platzer, Stefan Mitsch: verification of hybrid systems, KeyMaera
- Me: formal verification and Lean

Areas of overlap:

- Designing special-purpose logics for reasoning about dynamical systems and control theory.
- Implementing them / studying them in Lean.
- Verifying aspects of KeyMaera in Lean.
- Developing verification tools and backend automation in Lean.

Formal Methods in Analysis

The proposal was titled *Constructive Methods and Formal Verification in Analysis*.

We wanted a more snappy project name and acronym.

We settled on *Formal Methods in Analysis (FoMA)*.

I am using you all as consultants.

The Plan

- Tell you about the grant that supported this workshop.
- Tell you about Lean.
- Discuss plans for both.

The Lean Theorem Prover

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

Lean is open source, released under a permissive license, Apache 2.0.

See `http://leanprover.github.io`.

The Lean Theorem Prover

- based on a variant of the calculus of inductive constructions:
  - noncumulative universes
  - impredicative `Prop` with definitional proof irrelevance
  - Pi types
  - inductive type families with primitive recursors
  - quotient types
- additional axioms:
  - propositional extensionality
  - a choice operator for classical logic
- small kernel with three independent reference checkers
- well designed parser and elaborator
  - nice syntax
  - elaborator is fast and robust
  - type class inference

The Lean Theorem Prover

- virtual machine evaluator:
  - computable definitions are compiled to byte code
  - performant, pure functional programming language
  - syntactic support for monadic programming
  - native handling of nats, integers, strings, arrays
  - io (Lean's package manager is written in Lean)
  - profiler, debugger
- a powerful *metaprogramming* language:
  - a rich API to Lean internals
  - most Lean tactics are written in Lean
  - it is easy (and fun) to write more
- some automation:
  - simplifier
  - ematching
- nice user interaction
  - continuous compilation in Emacs or VSCode
  - Javascript version runs in Monaco in your browser

Logical Foundations

Lean is based on a version of the Calculus of Inductive Constructions, with:

- a hierarchy of (non-cumulative) universes, with a type *Prop* of propositions at the bottom
- dependent function types (Pi types)
- inductive types (à la Dybjer)

Semi-constructive axioms and constructions:

- quotient types (the existence of which imply function extensionality)
- propositional extensionality

A single classical axiom:

- choice

Logical Foundations

Lean has a hierarchy of non-cumulative type universes:

`Sort 0, Sort 1, Sort 2, Sort 3, ...`

These can also be written:

`Prop, Type 0, Type 1, Type 2, ...`

`Prop` is impredicative and definitionally proof irrelevant.

The latter means that if `p : Prop`, `s : p`, and `t : p`, then `s` and `t` are definitionally equal.

Logical Foundations

We have dependent function types `Π x : α, β x` with the usual reduction rule, `(λ x, t) s = t [s / x]`.

We have eta equivalence for functions: `t` and `λ x, t x` are definitionally equal.

We also have "let" definitions, `let x := s in t`, with the expected reduction rule.

Logical Foundations

Lean implements inductive families with primitive recursors, and the expected computation rules.

```
inductive vector (α : Type u) : ℕ → Type u
| nil : vector 0
| cons {n : ℕ} (a : α) (v : vector n) : vector (n+1)

#check (vector : Type u → ℕ → Type u)
#check (vector.nil : Π α : Type u, vector α 0)
#check (@vector.cons : Π {α : Type u} {n : ℕ},
  α → vector α n → vector α (n + 1))
#check (@vector.rec :
  Π {α : Type u} {C : Π (n : ℕ), vector α n → Sort u},
      C 0 (vector.nil α) →
      (Π {n : ℕ} (a : α) (v : vector α n), C n v →
                                  C (n + 1) (vector.cons a v)) →
      Π {n : ℕ} (v : vector α n), C n v)
```

Logical Foundations

We can quotient by an arbitrary binary relation:

```
constant quot :
  Π {α : Sort u}, (α → α → Prop) → Sort u
constant quot.mk :
  Π {α : Sort u} (r : α → α → Prop), α → quot r
axiom quot.ind :
  ∀ {α : Sort u} {r : α → α → Prop} {β : quot r → Prop},
    (∀ a, β (quot.mk r a)) → ∀ (q : quot r), β q
constant quot.lift :
  Π {α : Sort u} {r : α → α → Prop}
    {β : Sort u} (f : α → β),
    (∀ a b, r a b → f a = f b) → quot r → β
axiom quot.sound :
  ∀ {α : Type u} {r : α → α → Prop} {a b : α},
    r a b → quot.mk r a = quot.mk r b
```

These (with eta) imply function extensionality.

Logical Foundations

Propositional extensionality:

```
axiom propext {a b : Prop} : (a ↔ b) → a = b
```

Finally, we can go classical:

```
axiom choice {α : Sort u} : nonempty α → α
```

Here, `nonempty α` is equivalent to `∃ x : α, true`.

Diaconescu's trick gives us the law of the excluded middle.

Definitions that use choice to produce data are `noncomputable`.

Defining Functions

Lean's primitive recursors are a very basic form of computation.

To provide more flexible means of defining functions, Lean uses an *equation compiler*.

It does pattern matching:

```
def list_add {α : Type u} [has_add α] :
  list α → list α → list α
| [] _                := []
| _  []               := []
| (a :: l) (b :: m) := (a + b) :: list_add l m

#eval list_add [1, 2, 3] [4, 5, 6, 6, 9, 10]
```

Defining Functions

It handles arbitrary structural recursion:

```
def fib : ℕ → ℕ
| 0       := 1
| 1       := 1
| (n+2) := fib (n+1) + fib n

#eval fib 10
```

It detects impossible cases:

```
def vector_add [has_add α] :
  Π {n}, vector α n → vector α n → vector α n
| ._  nil              nil        := nil
| ._  (@cons ._ _ a v) (cons b w) := cons (a + b)
                                         (vector_add v w)

#eval vector_add (cons 1 (cons 2 (cons 3 nil)))
                 (cons 4 (cons 5 (cons 6 nil)))
```

Defining Inductive Types

Nested and mutual inductive types are also compiled down to the primitive versions:

```
mutual inductive even, odd
with even : ℕ → Prop
| even_zero : even 0
| even_succ : ∀ n, odd n → even (n + 1)
with odd : ℕ → Prop
| odd_succ : ∀ n, even n → odd (n + 1)

inductive tree (α : Type)
| mk : α → list tree → tree
```

Defining Functions

The equation compiler handles nested inductive definitions and mutual recursion:

```
inductive term
| const : string → term
| app   : string → list term → term

open term

mutual def num_consts, num_consts_lst
with num_consts : term → nat
| (term.const n)  := 1
| (term.app n ts) := num_consts_lst ts
with num_consts_lst : list term → nat
| []      := 0
| (t::ts) := num_consts t + num_consts_lst ts

def sample_term := app "f" [app "g" [const "x"], const "y"]

#eval num_consts sample_term
```

Defining Functions

We can do well-founded recursion:

```
def div : nat → nat → nat
| x y :=
  if h : 0 < y ∧ y ≤ x then
    have x - y < x, from sorry,
    div (x - y) y + 1
  else
    0
```

Here is Ackermann's function:

```
def ack : nat → nat → nat
| 0     y     := y+1
| (x+1) 0     := ack x 1
| (x+1) (y+1) := ack x (ack (x+1) y)
```

Defining Functions

Here is another example:

```
def nat_to_bin : ℕ → list ℕ
| 0       := [0]
| 1       := [1]
| (n + 2) :=
  have (n + 2) / 2 < n + 2, from sorry,
  (nat_to_bin ((n + 2) / 2)).concat (n % 2)

#eval nat_to_bin 1234567
```

Type Class Inference

Type class resolution is well integrated.

```
class semigroup (α : Type u) extends has_mul α :=
(mul_assoc : ∀ a b c, a * b * c = a * (b * c))

class monoid (α : Type u) extends semigroup α, has_one α :=
(one_mul : ∀ a, 1 * a = a) (mul_one : ∀ a, a * 1 = a)

def pow {α : Type u} [monoid α] (a : α) : ℕ → α
| 0     := 1
| (n+1) := a * pow n

theorem pow_add {α : Type u} [monoid α] (a : α) (m n : ℕ) :
  a^(m + n) = a^m * a^n :=
begin
  induction n with n ih,
  { simp [add_zero, pow_zero, mul_one] },
  rw [add_succ, pow_succ', ih, pow_succ', mul_assoc]
end

instance : linear_ordered_comm_ring int := ...
```

Syntactic Gadgets

We have a number of nice syntactic gadgets:

- Anonymous constructors and projections.
- Uniform notation for records.
- Pattern matching with assumptions and `let`.
- Monadic *do* notation.
- Default arguments, optional arguments, thunks, tactic handlers.

Syntactic Gadgets: Anonymous Constructors

```
example : p ∧ q → q ∧ p :=
λ h, and.intro (and.right h) (and.left h)

example : p ∧ q → q ∧ p :=
λ h, ⟨h.right, h.left⟩

#eval list.map (λ n, n * n) (range 10)

#eval (range 10).map $ λ n, n * n
```

Syntactic Gadgets: Record Notation

```
structure color :=
mk :: (red : nat := 0) (green : nat := 0) (blue : nat := 0)

#check color.mk 100 200 150
#check { color . red := 100, green := 200, blue := 203 }

-- you can omit the name of the structure when it can be inferred
def hot_pink : color := { red := 255, green := 192, blue := 203 }
def my_color : color := ⟨100, 200, 150⟩

-- defaults are used when omitted
example : { color . red := 100 }.blue = 0 := rfl

-- notation for record update
def lavender := { hot_pink with green := 20 }

#eval lavender.red
#eval lavender.green
```

Syntactic Gadgets: Pattern Matching

```
example : (∃ x, p x) → (∃ y, q y) → ∃ x y, p x ∧ q y
| ⟨x, px⟩ ⟨y, qy⟩ := ⟨x, y, px, qy⟩

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y) : ∃ x y, p x ∧ q y :=
match h₀, h₁ with
  ⟨x, px⟩, ⟨y, qy⟩ := ⟨x, y, px, qy⟩
end

example (h₀ : ∃ x, p x) (h₁ : ∃ y, q y) : ∃ x y, p x ∧ q y :=
let ⟨x, px⟩ := h₀,
    ⟨y, qy⟩ := h₁ in
⟨x, y, px, qy⟩

example : (∃ x, p x) → (∃ y, q y) → ∃ x y, p x ∧ q y :=
λ ⟨x, px⟩ ⟨y, qy⟩, ⟨x, y, px, qy⟩
```

Syntactic Gadgets: Monads

```
variables (l : list α) (f : α → list β)
variables (g : α → β → list γ) (h : α → β → γ → list δ)

example : list δ :=
do a ← l,
   b ← f a,
   c ← g a b,
   h a b c
```

Lean instantiates:

- the option monad
- the list monad
- the state monad
- a tactic state monad for metaprogramming
- monad transformers (especially: adding state)

Lean as a Programming Language

Lean implements a fast bytecode evaluator:

- It uses a stack-based virtual machine.
- It erases type information and propositional information.
- It uses eager evaluation (and supports delayed evaluation with thunks).
- You can use anything in the Lean library, as long as it is not `noncomputable`.
- The machine substitutes native nats and ints (and uses GMP for large ones).
- It substitutes a native representation of arrays.
- It has a profiler and a debugger.
- It is really fast.

Lean as a Programming Language

```
#eval 3 + 6 * 27
#eval if 2 < 7 then 9 else 12
#eval [1, 2, 3] ++ 4 :: [5, 6, 7]
#eval "hello " ++ "world"
#eval tt && (ff || tt)

def binom : ℕ → ℕ → ℕ
| _      0      := 1
| 0      (_+1)  := 0
| (n+1) (k+1)  := if k > n then 0
                  else if n = k then 1
                  else binom n k + binom n (k+1)

#eval (range 7).map $ λ n, (range (n+1)).map $ λ k, binom n k
```

Lean as a Programming Language

```
section sort
universe variable u
parameters {α : Type u} (r : α → α → Prop) [decidable_rel r]
local infix ≼ : 50 := r

def ordered_insert (a : α) : list α → list α
| []       := [a]
| (b :: l) := if a ≼ b then a :: (b :: l)
              else b :: ordered_insert l

def insertion_sort : list α → list α
| []       := []
| (b :: l) := ordered_insert b (insertion_sort l)

end sort

#eval insertion_sort (λ m n : ℕ, m ≤ n)
  [5, 27, 221, 95, 17, 43, 7, 2, 98, 567, 23, 12]
```

Lean as a Programming Language

There are algebraic structures that provides an interface to terminal and file I/O.

At some point, we decided we should have a package manager to manage libraries and dependencies.

Gabriel Ebner wrote one, in Lean.

Lean as a Metaprogramming Language

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

Various answers:

- Use the underlying implementation language (ML, OCaml, C++, . . .).
- Use a domain-specific tactic language (LTac, MTac, Eisbach, . . .).
- Use reflection (RTac).

Metaprogramming in Lean

Our answer: go meta, and use the object language.

(MTac, Idris, and now Agda do the same, with variations.)

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.

Metaprogramming in Lean

The method:

- Add an extra (meta) constant: `tactic_state`.
- Reflect expressions with an `expr` type.
- Add (meta) constants for operations which act on the tactic state and expressions.
- Have the virtual machine bind these to the internal representations.
- Use a tactic monad to support an imperative style.

Definitions which use these constants are clearly marked `meta`, but they otherwise look just like ordinary definitions.

Metaprogramming in Lean

```
meta def find : expr → list expr → tactic expr
| e []         := failed
| e (h :: hs) :=
  do t ← infer_type h,
     (unify e t >> return h) <|> find e hs

meta def assumption : tactic unit :=
do { ctx ← local_context,
     t   ← target,
     h   ← find t ctx,
     exact h }
<|> fail "assumption tactic failed"

lemma simple (p q : Prop) (h₁ : p) (h₂ : q) : q :=
by assumption
```

Lean 4

Leo de Moura and Sebastian Ullrich are developing Lean 4, in a private repository.

It will be made public when ready.

We shouldn't expect backward compatibility.

In the meanwhile, we are working with Lean 3.4.1, which will not change.

Lean 4

Goals:

- New compiler and C++ code generator.
- JIT compiler on top of LLVM.
- New runtime (support for unboxed values and FFI).
- New monad for accessing primitives that are currently available only in C++ (e.g., type-context).
- New parser and macro expander (in Lean).
- Make sure that tactics such as simp can be efficiently implemented in Lean.
- Fix critical issues (e.g., issue #1601).
- Fix language issues (e.g., parameters, kernel enforced private declarations, kernel support for nested inductive datatypes).
- Reduce clutter in the core lib and code base.

The Plan

- Tell you about the grant that supported this workshop.
- Tell you about Lean.
- Discuss plans for both.

Whither interactive theorem proving?

Interactive theorem proving is a hard sell.

For most mathematicians, fully verified mathematics is unattractive: it is unnecessary and too much work.

The same is true for most software developers, engineers, etc.

What would make interactive systems more useful and appealing?

Explorations with Lean

Other ways to interactive theorem provers:

- to write precise specifications
- to verify subtle parts of mathematical arguments
- to explore interactively
- to call external reasoning tools interactively
  - trusting some of them
  - reconstruct proofs
  - verifying certificates
- to combine results from different platforms
- to embed custom logics and reasoning procedures
- to write custom automation
- to search for results in a library
- to teach mathematics

Plans for FoMA

General approaches:

- Develop the libraries.
- Embed special-purpose logics.
- Use Lean as a front end to other solvers and provers.
- Develop custom decision procedures and automation.

Domains of application:

- Proving stability, robustness:
    - manipulate specifications
    - synthesize Lyapunov functions
    - synthesize invariants
    - import and verify certificates
- Optimization:
    - manipulate specifications
    - verify certificates

Lean as a front end

The model:

- Write a specification in Lean.
- Manipulate it using tactics.
- Send verification conditions to back-end provers.
- Verify results (when possible).
- Patch together within Lean.

Lean as a front end

Back end tools:

- SAT solvers
- SMT solvers
- computer algebra systems
- semidefinite optimization, convex algebraic geometry
- validated numerics
- symbolic decision procedures (RCF, ACF, linear / integer arithmetic)

Lean as a front end

The challenges:

- Develop convenient ways of manipulating specifications.
- Interface with external provers.
- Reconstruct proofs from certificates.
- Synthesize useful information, like invariants.
- Develop decision procedures.

Embedding logics in Lean

Question: how and to what extent can we embed special purpose logics in Lean?

- shallow embedding: roughly, an interpretation
- deep embedding: model the syntax, semantics

Lean's metaprogramming, custom tactic modes, and (eventually) programmable syntax should open up interesting opportunities.

Targets:

- modal logics
- dynamic logics (including differential dynamic logic)
- cohesion
- differential dynamic logic
- certificates: resolution, SMT proofs