Efficient verification and metaprogramming in Lean

Leonardo de Moura - Microsoft Research

22nd International Symposium on Formal Methods,
16 July 2018, Oxford UK

http://leanprover.github.io/

Microsoft

Lean aims to bridge the gap between interactive and automated theorem proving

Introduction: Lean

- Goals
  - Extensibility
  - Expressivity
  - Scalability
  - Proof stability

- Platform for
  - Developing custom automation and Domain Specific Languages
  - Software verification
  - Formalized Mathematics

- Dependent Type Theory

- de Bruijn's principle: small trusted kernel, and two external type checkers

  "Hack without fear"

Metaprogramming

- Extend Lean using Lean

- Access Lean internals using Lean
  - Type inference
  - Unifier
  - Simplifier
  - Decision procedures
  - Type class resolution
  - …

- Proof/Program synthesis

How are automated provers used at Microsoft?

Testing

SAGE

Software Verification

Alive Ivy

Theorem Prover

Software verification & automated provers

- Easy to use for simple properties
- Main problems:
  - Scalability issues
  - Proof stability
  - Hard to control the behavior of automated provers
- in many verification projects:
  - Hyper-V
  - Ironclad & Ironfleet (https://github.com/Microsoft/Ironclad)
  - Everest (https://project-everest.github.io/)

Automated provers are mostly blackboxes

"The Strategy Challenge in SMT Solving",
joint work with Grant Passmore

$$(\mathsf{check}(\neg \mathsf{diff} \vee \frac{\mathsf{atom}}{\mathsf{dim}} < \mathsf{k}) \,;\, \mathsf{simplex}) \,|\, \mathsf{floydwarshall}$$

$$\mathsf{simplify} \,;\, \mathsf{gaussian} \,;\, (\mathsf{modelfinder} \,|\, \mathsf{smt}(\mathsf{apcad}(\mathsf{icp})))$$

Solver is still an "end-game" strategy

Software verification tools are starting to use tactics

Who checks the VCGen?

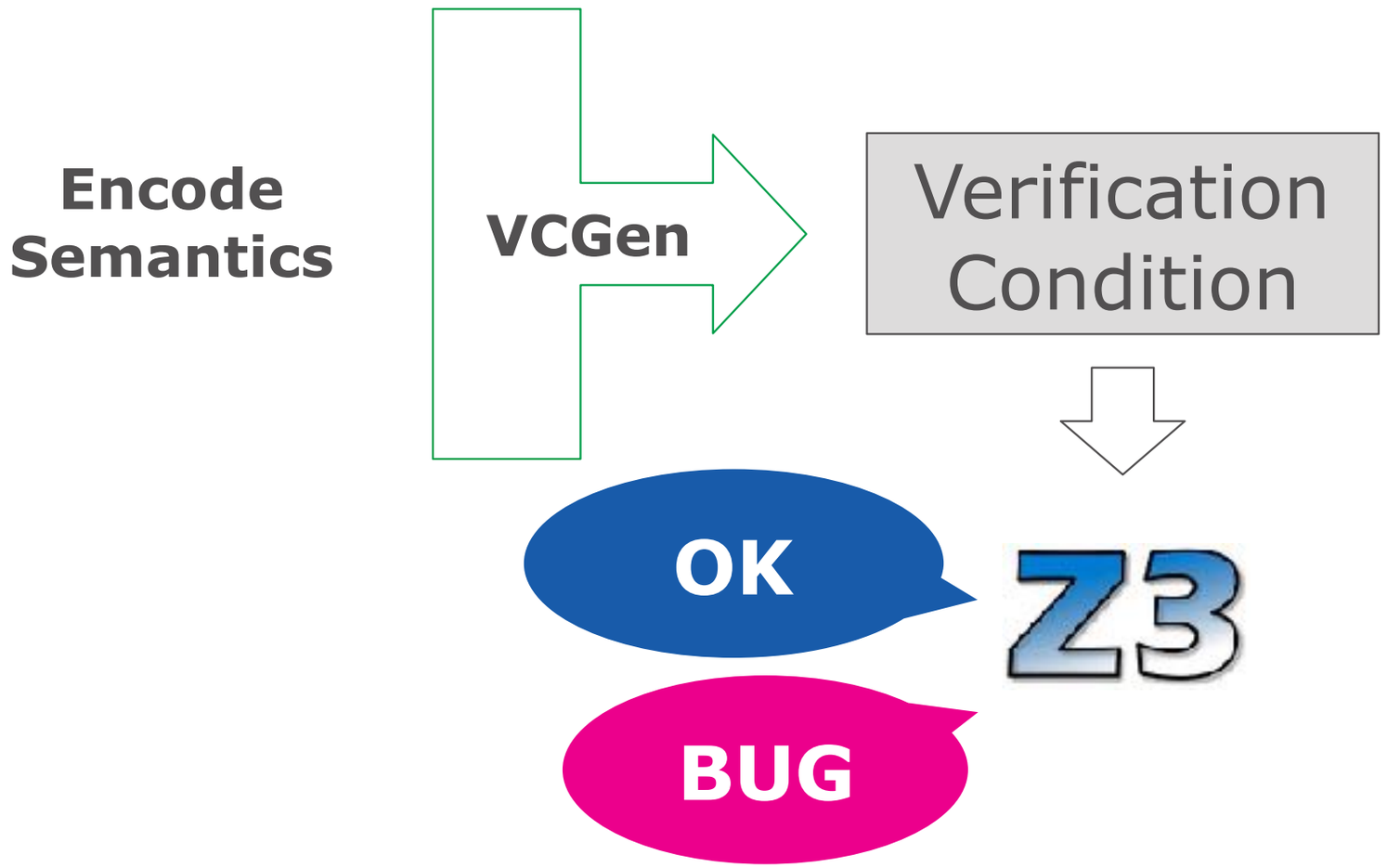
Alive (Nuno Lopes, MSR Cambridge)

```
Pre: isPowerOf2(C)
%s = shl C, %N
%q = zext %s
%r = udiv %x, %q

  =>

%N2 = add %N, log2(C)
%N3 = zext %N2
%r  = lshr %x, %N3
```

<Input>



Alive Bug

https://rise4fun.com/Alive/szp

```
Is this optimization correct?
    1 %a = shl %in, 1
    2 =>
    3 %a = add %in, %in
    4
```

home permalink
'►' shortcut: Alt+B

```
----------------------------------------
Optimization: 1
Done: 64
Optimization is correct!
```

“The bug in the VCGen was there for 4 years” Nuno Lopes

How serious is the problem?

- View Alive as a “bug finder”
  - Reasonable when peephole optimizations are written by humans
  - Alive is very successful
    - LLVM team uses Alive all the time
- Peephole optimizations are now being synthesized automatically
  - Example: Souper https://github.com/google/souper
- Make sure VCGen is correct when verifying synthesized code

Applications

AliveinLean

Re-implementation of Alive in Lean

Nuno Lopes (MSR Cambridge)

Open source: https://github.com/Microsoft/AliveInLean

Pending issues:

- Using processes+pipes to communicate with Z3
- Simpler framework for specifying LLVM instructions

IVy Metatheory

IVy is a tool for protocol verification

Ken McMillan, MSR Redmond

https://github.com/Microsoft/ivy/wiki/The-IVy-language

Certigrad

Bug-free machine learning on stochastic computation graphs
Daniel Selsam (Stanford)

Source code: https://github.com/dselsam/certigrad
ICML paper: https://arxiv.org/abs/1706.08605
Video: https://www.youtube.com/watch?v=-A1tVNTHUFw
Certigrad at Hacker news: https://news.ycombinator.com/item?id=14739491

Protocol verification

Joe Hendrix, Joey Dodds, Ben Sherman, Ledah Casburn, Simon Hudon
Galois inc

“We defined a hash-chained based distributed time stamping service down to the byte-level message wire format, and specified the system correctness as an LTL liveness property over an effectively infinite number of states, and then verified the property using Lean. **We used some custom tactics for proving the correctness of the byte-level serialization/deserialization routines**, defined an abstraction approach for reducing reasoning about the behavior of the overall network transition system to the behavior of individual components, and then verified those components primarily using existing Lean tactics.”

https://github.com/GaloisInc/lean-protocol-support

SQL query equivalence

Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, Dan Suciu
University of Washington
https://arxiv.org/pdf/1802.02229.pdf

Lean forward

Lean Forward

Usable Computer-Checked Proofs and Computations for Number Theorists

PI: Jasmin Blanchette (University of Amsterdam)

Sanders Damen, Johannes Hölzl, Robert Lewis, Assia Mahboubi, Freek Weidijk

https://lean-forward.github.io/

https://lean-forward.github.io/lean-together/2019/index.html

Formal Abstracts

Tom Hales (University of Pittsburgh)

“To develop software and services for transforming mathematical results as they appear in journal article abstracts into formally structured data that machines can read, process, search, check, compute with, and learn from as logical statements.”

https://sloan.org/grant-detail/8439

https://hanoifabs.wordpress.com/2018/05/31/tentative-schedule/

https://github.com/formalabstracts/formalabstracts

mathlib

Lean mathematical components library

Mario Carneiro (CMU)

Johannes Hölzl (University of Amsterdam)

https://github.com/leanprover/mathlib

Many contributors: https://github.com/leanprover/mathlib/graphs/contributors

https://leanprover.zulipchat.com/

Education

Logic and Proof, undergraduate course
Jeremy Avigad (CMU)
https://leanprover.github.io/logic_and_proof/introduction.html

Type theory, graduate course
Jeremy Avigad (CMU)
https://leanprover.github.io/theorem_proving_in_lean/introduction.html

Programming Languages, graduate course
Zack Tatlock (University of Washington)

Introduction to Proof, 1st year undergraduate course
Kevin Buzzard (Imperial College)

External perception

Lean

Lean is ambitious, and it will be massive. Do not be fooled by the name. "*Construction area keep out*" signs are prominently posted on the perimeter fencing.

From https://jiggerwit.wordpress.com/2018/04/14/the-architecture-of-proof-assistants/

Lean timeline

- Lean 1 (2013) Leo and Soonho Kong
  - Almost useless
  - Brave (crazy?) users in 2014: Jeremy Avigad, Cody Roux and Floris van Doorn
- Lean 2 (2015) Leo and Soonho Kong
  - First official release
  - Emacs interface
  - Floris van Doorn develops the HoTT library for Lean
  - Jakob von Raumer Master Thesis
  - Math library (Jeremy Avigad, Rob Lewis and many others)
- Lean 3 (2016) Leo, Daniel Selsam, Gabriel Ebner, Jared Roesch, Sebastian Ullrich
  - Lean is now a programming language
  - Meta programming
  - White box automation
  - VS Code interface

White-box automation

APIs (in Lean) for accessing data-structures and procedures found in SMT solvers and ATPs.

Example: congruence closure

```
/-- Congruence closure state -/
meta constant cc_state                  : Type
meta constant cc_state.mk_core          : cc_config → cc_state
/-- Create a congruence closure state object using the hypotheses in the current goal. -/
meta constant cc_state.mk_using_hs_core : cc_config → tactic cc_state
meta constant cc_state.next             : cc_state → expr → expr
meta constant cc_state.root             : cc_state → expr → expr
meta constant cc_state.mt               : cc_state → expr → nat
meta constant cc_state.internalize      : cc_state → expr → tactic cc_state
meta constant cc_state.add              : cc_state → expr → tactic cc_state
meta constant cc_state.is_eqv           : cc_state → expr → expr → tactic bool
meta constant cc_state.is_not_eqv       : cc_state → expr → expr → tactic bool
meta constant cc_state.eqv_proof        : cc_state → expr → expr → tactic expr
meta constant cc_state.inconsistent     : cc_state → bool
/-- `proof_for cc e` constructs a proof for e if it is equivalent to true in cc_state -/
meta constant cc_state.proof_for        : cc_state → expr → tactic expr
/-- `refutation_for cc e` constructs a proof for `not e` if it is equivalent to false in cc_state -/
meta constant cc_state.refutation_for   : cc_state → expr → tactic expr
/-- If the given state is inconsistent, return a proof for false. Otherwise fail. -/
meta constant cc_state.proof_for_false  : cc_state → tactic expr
```

Example: congruence closure

```
meta def fold_eqc_core {α} (s : cc_state) (f : α → expr → α) (first : expr) : expr → α → α
| c a :=
  let new_a := f a c,
      next  := s.next c in
  if next =ₐ first then new_a
  else fold_eqc_core next new_a

meta def fold_eqc {α} (s : cc_state) (e : expr) (a : α) (f : α → expr → α) : α :=
fold_eqc_core s f e e a
```

rsimp tactic

```
meta def collect_implied_eqs : tactic cc_state := ...

meta def choose (ccs : cc_state) (e : expr) : expr :=
ccs.fold_eqc e e $ λ (best_so_far curr : expr),
  if size curr < size best_so_far then curr else best_so_far
```

As in Haskell, the notation `f $ g t` is an alternative way of writing `f (g t)`

```
meta def rsimp : tactic unit :=
do ccs ← collect_implied_eqs,
   try $ simp_top_down $ λ t, do
      let root := ccs.root t,
      let t'   := choose ccs root,
      p        ← ccs.eqv_proof t t',
      return (t', p)
```

Extending the tactic state

```
def state_t (σ : Type) (m : Type → Type) [monad m] (α : Type) : Type :=
σ → m (α × σ)

meta constant smt_goal : Type
meta def smt_state  := list smt_goal
meta def smt_tactic := state_t smt_state tactic

meta def eblast : smt_tactic unit := repeat (ematch; try close)

meta def collect_implied_eqs : tactic cc_state :=
focus $ using_smt $ do
  add_lemmas_from_facts, eblast,
  (done; return cc_state.mk) <|> to_cc_state
```

Superposition prover

- 2200 lines of code

```
example {α} [monoid α] [has_inv α] : (∀ x : α, x * x⁻¹ = 1) →
                                     ∀ x : α, x⁻¹ * x = 1 :=
by super with mul_assoc mul_one
```

```
meta structure prover_state :=
(active passive : rb_map clause_id derived_clause)
(newly_derived : list derived_clause) (prec : list expr)
(locked : list locked_clause) (sat_solver : cdcl.state)
...
meta def prover := state_t prover_state tactic
```

dlist

```
structure dlist (α : Type u) :=
(apply     : list α → list α)
(invariant : ∀ l, apply l = apply [] ++ l)
```

```
def to_list : dlist α → list α
| ⟨xs, _⟩ := xs []
```

```
local notation `♯`:max := by abstract {intros, rsimp}
```

```
/-- `O(1)` Append dlists -/
protected def append : dlist α → dlist α → dlist α
| ⟨xs, h₁⟩ ⟨ys, h₂⟩ := ⟨xs ∘ ys, ♯⟩

instance : has_append (dlist α) :=
⟨dlist.append⟩
```

transfer tactic

- Developed by Johannes Hölzl (approx. 200 lines of code)

```
lemma to_list_append (l₁ l₂ : dlist α) : to_list (l₁ ++ l₂) = to_list l₁ ++ to_list l₂ :=
show to_list (dlist.append l₁ l₂) = to_list l₁ ++ to_list l₂, from
by cases l₁; cases l₂; simp; rsimp
```

```
protected def rel_dlist_list (d : dlist α) (l : list α) : Prop :=
to_list d = l
```

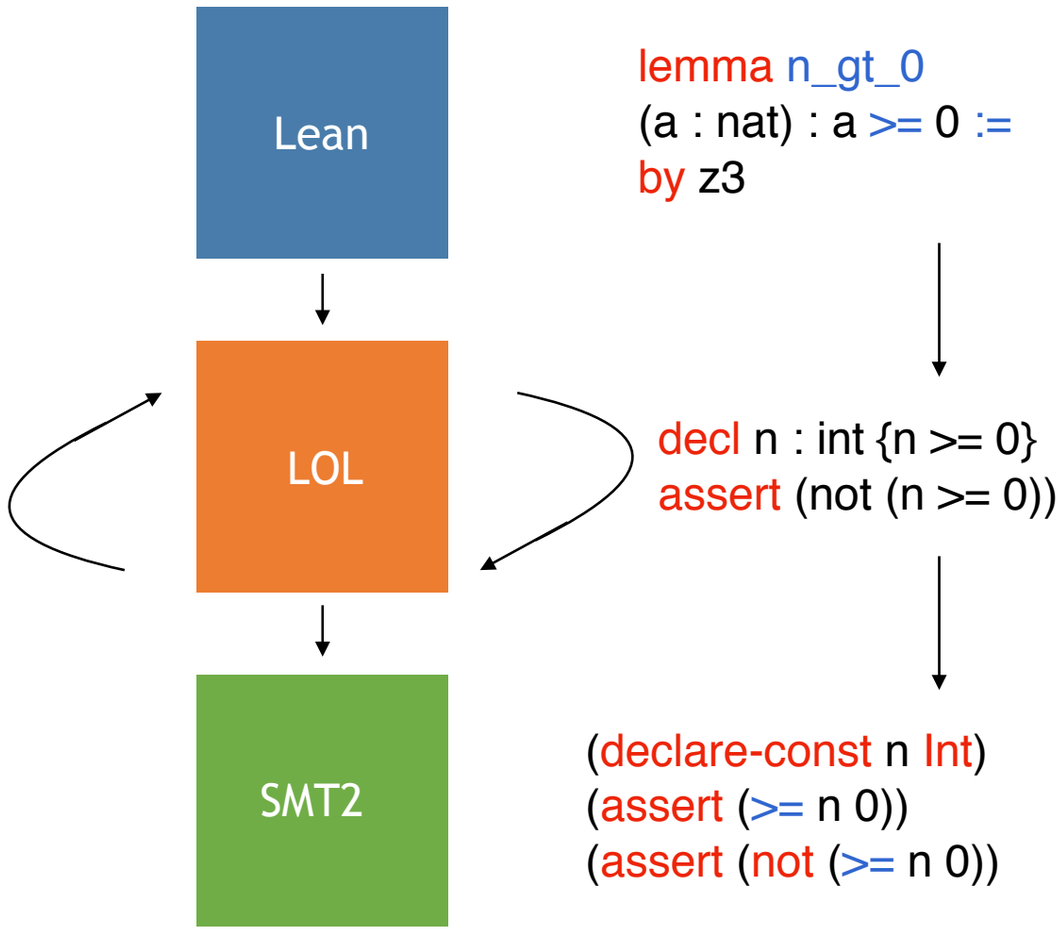
```
protected meta def transfer : tactic unit := do
  _root_.transfer.transfer [`relator.rel_forall_of_total, `dlist.rel_eq, `dlist.rel_empty,
    `dlist.rel_singleton, `dlist.rel_append, `dlist.rel_cons, `dlist.rel_concat]

example : ∀(a b c : dlist α), a ++ (b ++ c) = (a ++ b) ++ c :=
begin
  dlist.transfer,
  intros,
  simp
end
```

- We also use it to transfer results from nat to int.

Lean to SMT2

- Goal: translate a Lean local context, and goal into SMT2 query.
- Recognize fragment and translate to low-order logic (LOL).
- Logic supports some higher order features, is successively lowered to FOL, finally SMT2.



```
mutual inductive type, term
with type : Type
| bool : type
| int : type
| var : string → type
| fn : list type → type → type
| refinement : type → (string → term) → type
with term : Type
| apply : string → list term → term
| true : term
| false : term
| var : string → term
| equals : term → term → term
| …
| forallq : string → type → term → term
```

```
meta structure context :=
(type_decl : rb_map string type)
(decls : rb_map string decl)
(assertions : list term)
```

```
meta def reflect_prop_formula' : expr → smt2_m lol.term
| `(¬ %%P) := lol.term.not <$> (reflect_prop_formula' P)
| `(%%P = %%Q) := lol.term.equals <$>
                  (reflect_prop_formula' P) <*>
                  (reflect_prop_formula' Q)
| `(%%P ∧ %%Q) := lol.term.and <$>
                  (reflect_prop_formula' P) <*>
                  (reflect_prop_formula' Q)
| `(%%P ∨ %%Q) := lol.term.or <$>
                  (reflect_prop_formula' P) <*>
                  (reflect_prop_formula' Q)
| `(%%P < %%Q) := reflect_ordering lol.term.lt P Q
| …
| `(true) := return $ lol.term.true
| `(false) := return $ lol.term.false
| e := …
```

Coinductive predicates

- Developed by Johannes Hölzl (approx. 800 lines of code)
- Uses impredicativity of Prop
- No kernel extension is needed

```
coinductive all_stream {α : Type u} (s : set α) : stream α → Prop
| step {} : ∀{a : α} {ω : stream α}, a ∈ s → all_stream ω → all_stream (a :: ω)
```

```
coinductive alt_stream : stream bool → Prop
| tt_step : ∀{ω : stream bool}, alt_stream (ff :: ω) → alt_stream (tt :: ff :: ω)
| ff_step : ∀{ω : stream bool}, alt_stream (tt :: ω) → alt_stream (ff :: tt :: ω)
```

Ring solver

- Developed by Mario Carneiro (approx. 500 lines of code)
- https://github.com/leanprover/mathlib/blob/master/tactic/ring.lean
- ring2 uses computational reflection

```
import tactic.ring

theorem ex1 (a b c d : int) : (a + 0 + b) * (c + d) = b*d + c*b + a * c + d * a :=
by ring

theorem ex2 (α : Type) [comm_ring α] (a b c d : α)
    : (a + 0 + b) * (c + d) = b*d + c*b + a * c + d * a :=
by ring
```

Fourier-Motzkin elimination

- Linear arithmetic inequalities
- Developed at Galois inc
- https://github.com/GaloisInc/lean-protocol-support/tree/master/galois/arith

Lean 3.x limitations

- Lean programs are compiled into byte code
- Lean expressions are foreign objects in the Lean VM
- Very limited ways to extend the parser

```
infix >=        := ge
infix ≥         := ge
infix >         := gt
```

```
notation `∃` binders `, ` r:(scoped P, Exists P) := r
```

```
notation `[` l:(foldr `, ` (h t, list.cons h t) list.nil `]`) := l
```

- Users cannot implement their own elaboration strategies
- Users cannot extend the equation compiler (e.g., support for quotient types)

Lean 4

- Leo and Sebastian Ullrich (and soon Gabriel Ebner)
- Implement Lean in Lean
  - parser, elaborator, equation compiler, code generator, tactic framework and formatter
- New intermediate representation (defined in Lean) can be translated into C++ (and LLVM IR)
- Only runtime, kernel and basic primitives are implemented in C++
- Users may want to try to prove parts of the Lean code generator or implement their own kernel in Lean
- Foreign function interface (invoke external tools)

Lean 4

- Runtime has support for boxed and unboxed data
- Lean 4 expressions are implemented using the Lean runtime
- Runtime uses reference counting for GC and performs destructive updates when RC = 1 (i.e., object is not shared)

- We have support for low-level tricks used in SMT and ATP. Example: pointer equality

```
def use_ptr_eq {α : Type u} {a b : α}
               (c : unit -> {r : bool // a = b → r = tt})
               : {r : bool // a = b → r = tt} := c ()
```

Given `@use_ptr_eq _ a b c`, compiler generates

```
if (addr_of(a) == addr_of(b)) return true; else return c();
```

Conclusion

- Users can create their on automation, extend and customize Lean
- Domain specific automation
- Internal data structures and procedures are exposed to users (e.g., congruence closure)
- Whitebox automation
- VCGen verification and synthesized programs
- Lean 4 automation written in Lean will be much more efficient
- Lean 4 will be more extensible
- New application domains
  - Lean 4 as a more powerful Z3Py
  - Lean 4 as a platform for developing domain specific languages