Automation and Computation in the Lean Theorem Prover

Robert Y. Lewis[1] and Leonardo de Moura[2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
rlewis1@andrew.cmu.edu
[2] Microsoft Research, Redmond, WA, USA
leonardo@microsoft.com

Abstract

We describe some details of the Lean theorem prover, focusing on the aspects that make Lean a good environment for automation and AI. We also outline a novel automated procedure for proving inequalities over $\mathbb{R}$ that is currently being implemented.

The Lean theorem prover. The Lean theorem prover is a new proof environment being developed at Microsoft Research ([3, 4]). While somewhat similar to Coq [2] in foundation and syntax, Lean aims to improve on the status quo in a number of ways. In dependent type theory, complex and subtle techniques are needed to elaborate terms in an intuitive way. Elaborated proofs are checked by a small trusted kernel that does not contain a termination checker, fixpoint operator, or pattern matching. An independent reference checker, less than 2000 lines of Haskell code, checks the entire standard library in only a few minutes. Lean combines an extremely efficient elaboration process with a powerful type class inference mechanism. This setting allows for a clean, uniform development of the algebraic hierarchy and number structures, making the system conducive to generalized automated methods.

Indeed, Lean has been developed from the beginning with automation in mind. While we will treat Lean as an interactive proof environment in this document, the system can also be seen as a body of verified mathematical claims and an API for connecting them; in effect, it is a setting for automated tools to assemble proofs. Lean aims to bridge the gap – or blur the line, if one likes – between user-centric interactive proving and machine-centric automated proving.

Lean's standard kernel implements a proof-irrelevant and impredicative `Prop`. While the core library is designed constructively, classical axioms can be imported and used seamlessly, and some theories have been developed classically. Users can define terms that compute, and then reason about them classically. We believe that these features, and the uniform library development and tools described below, make Lean a promising environment in which to develop a hammer-style ATP system.

Type class inference. Lean allows any family of inductive types to be marked as a *type class*. A term that instantiates one such type can be marked as an *instance* of the type class. The elaborator can then synthesize terms of this type by searching through declared instances. This search is not simply a lookup table, since instances can depend on other instances. In the following example, square brackets instruct the elaborator to infer the term using type class inference. The proof in the final line is inferred as well.

```
inductive decidable [class] (p : Prop) : Type := ...
definition dec_and [instance] (p q : Prop) [Hp : decidable p] [Hq : decidable q] : decidable (p ∧ q) := ...
constant P : ℕ → Prop
axiom Pn [instance] : ∀ n, decidable (P n)
example : decidable (P 2 ∧ P 3 ∧ P 4) := _
  -- inferred: @dec_and (P 2) (P 3 ∧ P 4) (Pn 2) (@dec_and (P 3) (P 4) (Pn 3) (Pn 4))
```

The type class inference tool employs a λ-Prolog-style [7] recursive, backtracking search, and caches results.

Algebraic hierarchy. The algebraic hierarchy serves two purposes in a proof assistant. First, these structures are objects of study in their own right: users prove theorems about groups, vector spaces, or fields in the abstract without intended applications. Second, these structures are instantiated by the concrete number structures: $\mathbb{R}$ forms a field, and theorems proved about fields should also apply to $\mathbb{R}$.

As in similar systems, Lean's algebraic structures are defined as record types indexed over a base type.

```
structure monoid [class] (A : Type) extends semigroup A, has_one A :=
  (one_mul : ∀a, mul one a = a) (mul_one : ∀a, mul a one = a)
```

The `extends` syntax ensures that all data fields required for a semigroup are also required for a monoid, and automatically constructs a definition of the form

```
definition monoid.to.semigroup [instance] [H : monoid A] : semigroup A := ...
```

by finding the appropriate projections. When one structure A' extends another structure A, all theorems proved about types instantiating A will also apply to types instantiating A'. For instance, the associativity of multiplication in a monoid follows from the corresponding theorem about semigroups. No duplication or instantiation of theorems is necessary. Since projections reduce definitionally, when type class inference finds different terms witnessing the same property, they will be definitionally equal.

The effect of this development is an algebraic hierarchy that behaves exactly as a mathematician would expect: structures are cumulative, and when using a certain theorem or property, there is no need to remember at what point of the hierarchy that property entered.

Number structures. The number structures $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ are all defined and instantiate the appropriate algebraic structures. Once `int_is_comm_ring [instance] : comm_ring int` has been defined, all theorems applicable to commutative rings are immediately available for $\mathbb{Z}$. Importantly, this process happens via exactly the same mechanism as in the algebraic hierarchy. Automation designed for, say, ordered fields will not distinguish between $\mathbb{Q}$ and $\mathbb{R}$, making the system clean and uniform.

Lean's number structures have been developed constructively, through showing that $\mathbb{R}$ is an ordered ring. The remainder of the construction of $\mathbb{R}$ and $\mathbb{C}$ is classical.

Numerical computation. Binary numerals can be used in Lean in any structure that has 0, 1, and +.

```
definition bit0 {A : Type} [s : has_add A]  (a  : A)                : A := add a a
definition bit1 {A : Type} [s : has_one A] [t : has_add A] (a : A) : A := add (bit0 a) one
definition add2 (n : ℕ) : ℕ := n + 2
```

The type of a numeral is inferred from context. Here, 2 is shorthand for `bit0 nat nat_has_add (one nat nat_has_one)`, where `nat_has_add : has_add nat` and `nat_has_one : has_one nat` are found by type class inference. In structures with the appropriate properties of addition, multiplication, subtraction, and division, numeric computations can be performed efficiently by binary arithmetic algorithms.

Uniformity for automation. The uniform approach using type class inference is conducive to both shallow and deep automation. Treating numerals generally makes arithmetic work identically across structures: computations with + and − in $\mathbb{Z}$ are exactly the same as in $\mathbb{R}$ and as in any ring. This allows Lean to have a powerful and flexible simplifier. Type class inference makes it trivial for automated techniques to detect what algebraic structures are present in a goal and what methods may be applicable. Since the same general theorem – not separate instantiations of one general theorem – asserts the associativity of addition in $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, etc., machine learning techniques do not need to be trained separately on examples for each structure.

Congruence closure for ITT. Selsam and de Moura [9] have developed a congruence closure algorithm for dependent type theory. Congruence closure is a vital tool for many ATP systems, but previously had never been adapted for the expressive logic of ITT. Other systems apply congruence closure only on simply-typed fragments of the language, or rely on users to manually discharge hypotheses. Selsam and de Moura describe an algorithm that captures the subtleties of congruence in ITT – assuming uniqueness of identity proofs – and implement the procedure in Lean.

Verifying non-linear inequalities. In interactive theorem proving, one often wishes to discharge simple arithmetic hypotheses that follow from premises in the context. Proof assistants implement solvers over various domains, for example linear constraints over $\mathbb{Q}$ or $\mathbb{Z}$. Higher-powered methods – e.g. CAD methods

for RCFs – can only generate proofs with much effort [6]. A class of relatively simple but highly nonlinear problems over $\mathbb{R}$ has proven very difficult to attack automatically. These problems show up often in mathematical analysis and in the verification of physical systems. For example, consider the inference

$$0 < x < y,\ u < v \ \Rightarrow\ 2u + \mathtt{exp}(1 + x + x^4) < 2v + \mathtt{exp}(1 + y + y^4).$$

This inference is tight, nonlinear, outside the theory of RCF, and difficult to capture systematically with backchaining methods; nevertheless, it looks "simple" to any mathematician. In [1, 5], Avigad, Lewis, and Roux describe a system, based on a Nelson-Oppen style architecture [8], that solves many such problems by saturating arithmetic facts and heuristically instantiating lemmas. While not a complete decision procedure, a prototype has been shown to perform well on real-life examples from ITP and system verification.

This system – nicknamed Polya – is currently being implemented as a tactic in Lean. One requirement of this system is that terms be rewritten to a normal form; Lean's flexible rewriter allows us to achieve this with minimal overhead. Polya makes many simple numerical calculations over $\mathbb{Q}$, which is efficient to verify in Lean. Once Polya is combined with a linear solver and general-purpose AI methods, we believe Lean will support a suite of automated tools exceeding that of other theorem provers.

References

[1] Jeremy Avigad, Robert Y. Lewis, and Cody Roux. A heuristic prover for real inequalities. *Journal of Automated Reasoning*, 56(3):367–386, 2016.

[2] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: The calculus of inductive constructions*. Springer-Verlag, Berlin, 2004.

[3] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *http://arxiv.org/pdf/1505.04324v1.pdf*, 2014.

[4] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). *http://leanprover.github.io/files/system.pdf*, 2014.

[5] Robert Y. Lewis. Polya: a heuristic procedure for reasoning with real inequalities. *MS Thesis, Dept. of Philosophy, Carnegie Mellon University*, 2014.

[6] Assia Mahboubi. Programming and certifying a CAD algorithm inside the Coq system. *Mathematics, Algorithms, Proofs*, 2006.

[7] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[8] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions of Programming Languages and Systems*, 1:245–257, 1979.

[9] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. *IJCAR*, 2016.