Interactive Theorem Proving with Lean

Cody Roux
Draper Labs

November 25, 2015

What is theorem proving?

The field of Theorem Proving investigates expressing and proving **mathematical theorems** with the help of computers

What is it good for?

Any mathematical theorem is game, but one of the prime applications is verification of **hardware and software applications.**

What is it good for... in CS?

"If you're dealing with life and death, consider proving some key algorithms. But in the real world, you arent gonna need it." – user RonJeffries, c2.com

"[T]he absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage." – De Millo, Lipton and Perlis, *Social Processes and Proofs of Theorems and Programs (1979)*

"Unfortunately, there is a wealth of evidence that automated verifying systems are out of the question" – *ibid*

"The amount of formal reasoning required to prove even the smallest programs is beyond the capacity of most of us." – anonymous user, c2.com

What is it good for... in CS?

But...

"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users." – Yang *et al., Finding and Understanding Bugs in C Compilers*

Some Milestones

In Math:

- The Four Color Theorem (2005, Coq) **every map can be colored with at most four colors**
- The Prime Number Theorem (2005, Isabelle) **the prime numbers grow like** $n/\log(n)$
- The Feit-Thompson Theorem (2012, Coq) **every group of odd order is solvable**
- The Kepler Conjecture (2014, HOL) **we know how to pack oranges**

In CS:

- Verification of the driverless Metro 14 line in Paris (1998, B-method)
- Verification of the CompCert C compiler (2009, Coq)
- Verification of the seL4 microkernel (2014, Isabelle)
- Many more!

But what about me?

Alright, enough sales pitch!

How can you get started with theorem proving?

Lean

Ok, some more sales pitch:

In this talk, I'll be using a new theorem prover called Lean, developed by

- Leo de Moura, at Microsoft Research
- Jeremy Avigad, Soonho Kong, Floris van Doorn at CMU
- And many others!

Can be found at `http://leanprover.github.io`

The core components

An Interactive Theorem Prover (ITP) is

- A theorem prover:

  It needs to be able to **verify** fully detailed formal proofs of theorems. We call this the **Kernel**

- Interactive

  It needs to help the user provide such proofs. This is called the **Elaborator** and **Proof Engine**.

The core components

Two silly examples:

The ideal prover

- The kernel: Some implementation of a powerful logic that can express all abstractions.
- The proof engine: Just a `Prove` button!

The simplest prover

- The kernel: some implementation of a simple logic, like First Order Logic.
- The engine: just a proof parser.

Reality

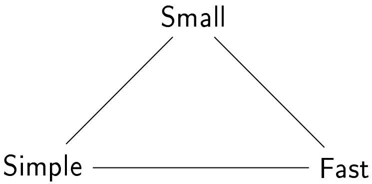
Somewhere in between.

The Kernel

The kernel is the most important **trusted component**. We need

- A simple but expressive logic
- A trustworthy implementation

The ideal implementation:



Choose 2!

The logic

We want simple (trustworthy, easy to implement) but powerful enough to formalize anything.

We use

- A theory of **dependent types** that unifies **function spaces, universal quantifiers and polymorphism**
- An impredicative type of propositions
- A hierarchy of **universes**
- Inductive families (GADTs in Haskell)
- Quotient types (useful to build $\mathbb{Z}$)

This is similar to the logic of Coq.

The implementation

- Simple
- Very Fast
- In C++
- $\sim$ 6000 lines of code

There is an alternate implementation of the kernel, in Haskell (a couple thousand lines).

A specification example

Let's work through an example to demonstrate the specification language:

Let's formalize the statement

There is an infinite number of primes.

A specification example

First observation: logical connectives can be represented by inductive types

```
inductive and (a b : Prop) : Prop :=
  intro : a → b → and a b

inductive or (a b : Prop) : Prop :=
| inl {} : a → or a b
| inr {} : b → or a b

inductive ⊤ : Prop := trivial : true

inductive ⊥ : Prop

definition ¬ (a : Prop) : Prop := a → ⊥
```

A specification example

Even existence and equality

```
inductive Exists {A : Type} (P : A → Prop) : Prop :=
    intro : ∀ (a : A), P a → Exists P
```

```
inductive eq {A : Type} (a : A) : A → Prop :=
    refl : eq a a
```

A specification example

Natural numbers are also specified by induction

```
inductive ℕ :=
| zero : ℕ
| succ : ℕ → ℕ
```

And operations are defined by recursion

```
definition add : ℕ → ℕ → ℕ
| add 0 m := m
| add (succ n’) m := succ (add n’ m)

definition mul : ℕ → ℕ → ℕ
| mul 0 m := m
| mul (succ n’) m := add m (mul n’ m)
```

A specification example

with a few notations

```
notation n `+` m := add n m
notation n `*` m := mul n m
...
```

```
definition le (n m : ℕ) := ∃ k, n + k = m

definition dvd (n m : ℕ) := ∃ k, n * k = m

definition prime (p : ℕ) :=
∀ d, 2 ≤ p ∧ (d | p → d = 1 ∨ d = p)
```

We can then express (and prove)

```
theorem inf_primes := ∀ n, ∃ p, n ≤ p ∧ prime p
```

The elaborator

To have any hope of being able to actually specify and prove anything, we need

1. a synthesizer for implicit arguments

```
eq.subst : ∀ {A : Type}{a b : A}{P : A → Prop},
      a = b → P a → P b
```

   we need to **synthesize** (guess) values for A, a, b and P!

2. a language for proofs

The elaborator

This problem is non-obvious:

```
eq.subst : ∀ {A : Type}{a b : A}{P : A → Prop},
     a = b → P a → P b
```

Say $P\ a \ \equiv\ a + a = 2$. Then we can have

$$P \ \equiv\ \lambda x.\ x + x = 2$$

or

$$P \ \equiv\ \lambda x.\ x + a = 2$$

$$P \ \equiv\ \lambda x.\ a + a = 2$$

we need to account and search for **multiple solutions**.

The proof language

How to design an effective high-level proof language is one of the major open questions in ITP research.

Right now there are 2 main approaches:

- A "stack based" **tactic language**, which operates by transforming the goal in a series of composable steps. Very powerful, but hard to read
- A declarative "isar-like" language, which is more verbose but easier to read and understand

On top of this, we need some powerful proof automation to help with the simpler intermediate steps.

Let's see some code!

Let's work through some simple examples.

Now is a good time for

Questions

A little more depth...

Some additional features:

- Implicit coercions
- Type Classes and "structures"
- Complex notations (infix, postfix, mixfix,...)
- Namespace management: local scopes, notations, coercions
- Lua bindings
- Javascript front end
- Quotient types
- A HoTT mode

Coercions

An important part of elaboration is the ability to insert **implicit casts**

```
variable {A : Type}
definition to_list [coercion] :
               ∀ {n : nat}, vector A n → list A
| to_list 0 nil := nil
| to_list (n+1) (a::v) := a::(to_list v)
variable f : list nat → Prop
variable v : vector nat 10
check f v -- f (@to_list nat 10 v)
```

Classes and structures

The notion of **dependent record** is a really natural way of representing **mathematical structures**

```
structure group (A : Type) :=
(mul : A → A → A)
(mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))
(...)
```

But it's better to use classes and inheritance for modularity

```
structure has_mul [class] (A : Type) :=
   (mul : A → A → A)
structure semigroup [class] (A : Type) extends has_mul A
   :=
(mul_assoc : ∀ a b c, mul (mul a b) c = mul a (mul b c))
 ...
```

Decidable Type Class

An element of Propis said to be decidable if we can decide whether it is true or false.

```
inductive decidable [class] (p : Prop) : Type :=
| inl : p → decidable p
| inr : ¬p → decidable p
```

Having an element `t : decidable p` is stronger than having an element `t : p ∨ ¬p`

The expression `if c then t else e` contains an implicit argument `[d : decidable c]`. If Hilbert's choice is imported, then all propositions are decidable (smooth transition to classical reasoning).

Quotient types

We support **quotient types**

- For any type A that supports an equivalence relation R we can form a type A/R
- A/R has the **universal property**: there is a function

$$f : A/R \to B$$

exactly when there is a $g : A \to B$ such that

$$\forall xy,\ x\ R\ y\ \Rightarrow g(x) = g(y)$$

HoTT

There has been a lot of buzz around the recent **synthetic approach to homotopy theory** based on type theory.

There is a mode in Lean designed for this. Some features

- No impredicative type `Prop`
- A special case of **higher inductive types** (n-truncations)

Libraries

A theorem prover is only as good as its libraries. Lean already has several

- `init`: the basic definitions of logical connectives, quantifiers, equality and $\mathbb{N}$
- `data`: All the programmers favorites: lists, ints, bools, finite sets and real numbers
- `algebra`: the hierarchy of algebraic structures, including groups, rings and fields
- `theories`: more fleshed out logical theories, with group theory (up to Sylow's theorems!), combinatorics, basic analysis and number theory

What about Coq?

The most obviously similar ITP in the same design space is Coq.

The major differences

- Similar core theories: dependent types, impredicative, universes
- Some additional features in Lean: Quotient types, HoTT mode
- Lean has recursors instead of built-in pattern matching (harder to get wrong)
- Less automation (for now)
- A mechanism for trust: you can call un-trusted decision procedures, which can be run in “safe mode”: the proof is then completely re-built