Dependent types and the algebraic hierarchy

Robert Y. Lewis

Carnegie Mellon University

June 19, 2015

Talk outline

Three goals for this talk:

- Explain the background of interactive theorem proving and dependent type theory
- Sell the proof assistant we've been working on
- Show off an interesting feature of this proof assistant and how we've made use of it

Why formal verification?

There are plenty of examples in mathematics and engineering where "human" certainty seems insufficient.

- Four-color theorem, Kepler conjecture, etc.
- Computer algebra systems (e.g. Mathematica)
- Intel processor verification

Why formal verification?

People have begun to use formal tools to verify proofs, computations, and software.

The high-level picture: users write “code” that approximates a proof. Computers take this code and generate a full proof certificate.

This is sort of like an axiomatic/natural deduction proof from logic class, but (hopefully!) less tedious, more comprehensive, less chance of human error.

Proof assistant ideals

A proof assistant is ideally:

- based on a familiar logic
- expressive enough for standard mathematics
- syntactically similar to informal math
- as automated as possible

Example

```
theorem one_lt_div_iff_lt (Hb : b > 0) : 1 < a / b ↔ b < a
:=
  have Hb' : b ≠ 0, from ne.symm (ne_of_lt Hb),
  iff.intro
    (assume H : 1 < a / b,
      calc
        b  < b * (a / b) : lt_mul_of_gt_one_right Hb H
        ... = a           : mul_div_cancel' Hb')
    (assume H : b < a,
     have Hbinv : 1 / b > 0, from  div_pos_of_pos Hb,
      calc
          1 = b * (1 / b) : mul_one_div_cancel Hb'
        ... < a * (1 / b) : mul_lt_mul_of_pos_right H Hbinv
        ... = a / b       : div_eq_mul_one_div)
```

Examples of proof assistants

- Mizar (1973): Tarski-Grothendieck set theory
- HOL family (1988): simple type theory
- Isabelle (1989): simple type theory
- Coq (1989): constructive dependent type theory
- PVS (1992): classical dependent type theory
- ACL2 (1996): primitive recursive arithmetic
- Agda (2007): constructive dependent type theory

  ...
- Lean (2013): constructive dependent type theory

Talk outline

A rough plan for the rest of this talk:

- A (quick!) overview of dependent type theory
- An introduction to Lean and its syntax
- Type class inference in Lean
- The Lean library and the algebraic hierarchy

Intro to DTT

Dependent type theory extends "simple" type theory (which extends the untyped λ calculus). As in the simple case:

- Every term has a type. `a : A`
- Given two types `A`, `B`, can construct product types $\texttt{A} \times \texttt{B}$ and function types $\texttt{A} \to \texttt{B}$
- Lambda expressions create terms of function types.
  $(\lambda\ \texttt{a : A, a}) : \texttt{A} \to \texttt{A}$

This simple type theory corresponds roughly to the $\wedge \to$ fragment of propositional logic.

Type constructors depend on other types, but not on terms.

Intro to DTT

In dependent type theory, this restriction is relaxed: types can take terms of other types as parameters.

`vector : (Π A : Type, Π n : ℕ, Type)`

The type of the output of a function can depend on the input:

`(λ A : Type, λ a : A, [a, a, a]) : vector A 3`

We can also create dependent pairs, where the type of the second term depends on the first term:

`(n : ℕ, [0, 1, ..., n]) : Σ n : ℕ, vector ℕ n`

Inductive types

Many concrete types we'll use are instances of inductive types:

```
inductive foo : Type :=
 | constructor_1 : ... → foo
 | constructor_2 : ... → foo
 ...
 | constructor_n : ... → foo
```

Each constructor specifies a way of building a term of type foo (possibly recursively). Every term of type foo has been constructed in one of these ways.

Functions can be defined on an inductive type using its recursor.

Curry-Howard Isomorphism

We can build a substantial amount of mathematics using nothing other than type universes, Π types, and inductive types.

In this framework, reasoning about propositions is basically the same as reasoning about data.

- $\Pi \sim \forall$
- $\Sigma \sim \exists$

The Lean theorem prover

Lean is a new theorem prover developed by Leonardo de Moura at Microsoft Research. It is based on dependent type theory.

We think of Lean as:

- An interactive theorem prover with powerful automation.
- An automated reasoning tool that produces proofs, has a rich language, can be used interactively, and is built on a verified mathematical library.

Lean's default logical framework is a version of the Calculus of Constructions with:

- an impredicative, proof-irrelevant type Prop of propositions
- a non-cumulative hierarchy of universes, Type 1, Type 2, . . . above Prop
- universe polymorphism
- inductively defined types

Features:

- The core is constructive.
- Can comfortably import classical logic.
- Can work in homotopy type theory.

Structures in Lean

In mathematics we prove theorems about general structures, and instantiate these structures. (Groups, rings, fields, ...)

A proof assistant must support this paradigm. If I prove that in an ordered field, $a > 0 \to 1/a > 0$, and show that $\mathbb{R}$ is an ordered field, I should be able to apply this theorem to $\mathbb{R}$.

The need for type inference

Consider the following mathematical statements:

"For every $x \in \mathbb{R}$, $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$."

"If G and H are groups and f is a homomorphism from G to H, then for every $a, b \in G$, $f(a \cdot b) = f(a) \cdot f(b)$."

"If F is a field of characteristic p and $a, b \in F$, then $(a + b)^p = \sum_{i=0}^{p} \binom{p}{i} a^i b^{p-i} = a^p + b^p$."

How do we parse these?

Structures in Lean

A structure is a non recursive inductive type with only one constructor.

The product and dependent product types are both examples of this.

Algebraic structures are examples of this: if `A` : `Type`, then `group A` is an inductive type whose sole constructor takes as arguments

- a function `mul` : `A` → `A` → `A`
- a function `inv` : `A` → `A`
- an object `one` : `A`
- proofs that `mul` is associative, `one` is the multiplicative identity, and `inv` is a left inverse

Structures in Lean

A theorem about ordered fields in general:

```
structure ordered_field (A : Type) := ...

theorem div_pos_of_pos (A : Type) (s : ordered_field A)
        (a : A) (H : 0 < a) : 0 < 1 / a :=
    lt_of_mul_lt_mul_left
      (mul_zero_lt_mul_inv_of_pos H)
      (le_of_lt H)
```

Specialized to $\mathbb{R}$:

```
definition reals_ordered_field : ordered_field ℝ := ...

(div_pos_of_pos ℝ reals_ordered_field) :
     ∀ a : ℝ, 0 < a → 0 < 1 / a
```

Two problems

```
(div_pos_of_pos ℝ reals_ordered_field) :
     ∀ a : ℝ, 0 < a → 0 < 1 / a
```

Problem 1: this description isn't complete. What are 0, 1, and / ?

- They are `ordered_field.zero reals_ordered_field`, `ordered_field.one reals_ordered_field`, `ordered_field.div reals_ordered_field`

Problem 2: it's annoying to reference the proof `reals_ordered_field` every time we use `div_pos_of_pos`.

The solution

Lean's solution to these problems is to use type class inference.

- Declare a (family of) inductive type(s) to be a type class.
- Declare instances of the type class.
- Mark some arguments with [ ] to denote that these arguments should be inferred.

Type class inference

An example of type class inference:

```
inductive inhabited [class] (A : Type) : Type :=
  mk : A → inhabited A

definition bool.is_inhabited [instance] : inhabited bool :=
  inhabited.mk bool.true

definition real.is_inhabited [instance] : inhabited real :=
  inhabited.mk real.one

definition default (A : Type) [H : inhabited A] : A :=
  inhabited.rec (λ a : A, a) H

check default bool -- bool
eval default real -- real.one
```

Chaining instances

```
definition prod.is_inhabited [instance] {A B : Type}
        [H1 : inhabited A] [H2 : inhabited B] :
        inhabited (A × B) :=
  inhabited.mk ((default A, default B))

eval default (real × real) -- (real.one, real.one)
eval default (real × bool × real) -- (real.one, bool.true,
    real.one)
```

This is accomplished by a recursive, backtracking search through declared instances.

Type classes in algebra

The algebraic example becomes:

```
structure ordered_field [class] (A : Type) := ...

theorem div_pos_of_pos {A : Type} [s : ordered_field A]
        (a : A) (H : 0 < a) : 0 < 1 / a :=
    lt_of_mul_lt_mul_left
      (mul_zero_lt_mul_inv_of_pos H)
      (le_of_lt H)

definition reals_ordered_field [instance] : ordered_field ℝ
    := ...

div_pos_of_pos :
  ∀ a : ℝ, 0 < a → 0 < 1 / a
```

Type classes in algebra

```
theorem div_pos_of_pos {A : Type} [s : ordered_field A]
        (a : A) (H : 0 < a) : 0 < 1 / a := ...
```

The < here is notation for `has_lt.lt {A : Type} [s : has_lt A]`.

Lean infers `has_lt` $\mathbb{R}$ from the chain

```
definition ordered_field.to_linear_order_pair [instance]
  {A : Type} [s : ordered_field A] : linear_order_pair A :=
    ...
definition linear_order_pair.to_order_pair [instance]
  {A : Type} [s : linear_order_pair A] : order_pair A :=
   ...
definition order_pair.to_has_lt [instance] {A : Type}
  [s : order_pair A] : has_lt A := ...
```

Notation overloading

This mechanism also lets us overload notation like <.

```
definition real.has_lt [instance] : has_lt real :=
  has_lt.mk real.lt
definition nat.has_lt [instance] : has_lt nat :=
  has_lt.mk nat.lt

check (λ a b : real, a < b) -- real → real → Prop
check (λ a b : nat, a < b) -- nat → nat → Prop
```

All of this applies to other operations: $+, *, \leq$, etc.

Type classes in algebra

The moral: because of the way algebraic structures extend and project down to each other, it's easy for type class inference to find the appropriate level to instantiate a particular theorem.

ℝ forms an ordered ring

```
theorem s_le.refl {s : reg_seq} : s_le s s :=
  begin
    let Hs := reg_seq.is_reg s,
    apply nonneg_of_nonneg_equiv,
    rotate 2,
    apply equiv.symm,
    apply neg_s_cancel s Hs,
    apply zero_nonneg,
    apply zero_is_reg,
    apply reg_add_reg Hs (reg_neg_reg Hs)
  end

theorem le.refl (x : ℝ) : x ≤ x :=
  quot.induction_on x (λ t, s.r_le.refl t)
```

ℝ forms an ordered ring

```
definition ordered_ring [instance] : algebra.ordered_ring ℝ
    :=
   {{algebra.ordered_ring, comm_ring,
    le_refl := le.refl,
    le_trans := le.trans,
    mul_pos := mul_gt_zero_of_gt_zero,
    mul_nonneg := mul_ge_zero_of_ge_zero,
    zero_ne_one := zero_ne_one,
    add_le_add_left := add_le_add_of_le_right,
    le_antisymm := eq_of_le_of_ge,
    lt_irrefl := not_lt_self,
    lt_of_le_of_lt := lt_of_le_of_lt,
    lt_of_lt_of_le := lt_of_lt_of_le,
    le_of_lt := le_of_lt,
    add_lt_add_left := add_lt_add_left}}
```

Decidable propositions

One more example of type class inference:

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

definition decidable_and [instance] (p q : Prop)
   [Hp : decidable p] [Hq : decidable q] :
   decidable (p ∧ q) := ...
definition decidable_or [instance] ...
definition decidable_implies [instance] ...

definition nat.lt.decidable [instance] (a b : nat) :
   decidable (a < b) := ...
```

Decidable propositions

```
eval (if (0 < 2 ∧ 2 < 5) ∨ (1 < 2 → 9 < 3) then 0 else 1)
    -- 0

theorem lt_next : 0 < 1 ∧ 1 < 2 ∧ 2 < 3 := dec_trivial
```