Introduction to Theorem Proving in Lean

Curry-Howard Isomorphism, Dependent Types and all that Jazz

Slavomir Kaslev
slavomir.kaslev@gmail.com

November 2, 2018

QOTD

"Point of view is worth 80 IQ points." Alan Kay

A Rather Unconventional Point of View

$$\text{Programming}^1 = \text{Mathematics}$$

Short History of Type Theory

- Types were first proposed by Bertrand Russell in 1902 to resolve Russell's paradox in Gottlob Frege's naive set theory
- Since then types have been studied on their own and combined with Alonzo Church's λ-calculus provide the theoretical framework of modern functional languages
- There are quite a few flavors of type theories around
  - Simply Typed Lambda Calculus (1940s), System F (1970s)
    - Haskell, OCaml, ML, ...
  - Martin-Löf Dependent Type Theory (1970s)
    - Agda, Coq, F*, Idris, Lean, ...
  - Homotopy Type Theory (2000s), Cubical Type Theory (2010s)
    - Agda with cubical paths, cubicaltt, yacctt, ...
- Today, while the OOP world is catching up with functional programming, Haskell is catching up with dependent types

Lean: An Open Source Theorem Prover

- Lean[2] is an open source theorem prover and programming language developed at Microsoft Research
- The project was launched in 2013 by Leonardo de Moura and is developed with the help of the community
    - Before Lean, Leonardo created the Z3[3] SMT solver
    - SMT solvers are fully automatic theorem provers — one specifies a problem in a representation that the solver understands and the solver will answer with either: *sat/unsat/unknown* along with a proof
    - In this sense, Lean is a semi-automatic theorem prover — one can either prove propositions manually, use automation with the help of tactics, or even interface with an external solver such as Z3
    - In contrast to several other theorem provers, Lean's tactics are implemented in Lean

---

[2]https://leanprover.github.io
[3]https://z3prover.github.io

Types: The Shape of Data

- Roughly speaking, types are a specification of their possible values
- In typed programming languages each valid expression a has some type A which we denote as $a : A$
- We can combine types to form new types, for example given two types A and B, we can create the types:
  - $A \rightarrow B$, the type of functions from A to B
  - $A \oplus B$, disjoint union of A and B
  - $A \otimes B$, product of A and B

Product and Sum Types as Inductive Types in Lean

```
inductive prod (A B : Type)
| mk (a : A) (b : B) : prod

inductive sum (A B : Type)
| inl {} (a : A) : sum
| inr {} (b : B) : sum
```

Inductive Types: Examples

```
inductive unit
| star : unit

inductive empty

inductive bool
| ff : bool
| tt : bool

inductive nat
| zero : nat
| succ (n : nat) : nat

inductive list (T : Type)
| nil {} : list
| cons (hd : T) (tl : list) : list

inductive bin_tree (T : Type)
| leaf (a : T) : bin_tree
| branch (left : bin_tree) (right : bin_tree) : bin_tree

inductive rose_tree (T : Type)
| node (a : T) (children : list rose_tree) : rose_tree

structure rose_tree (T : Type)
(a : T) (children : list (rose_tree T))
```

The Curry-Howard Correspondence: Propositions as Types

| Types	Logic
A	proposition
$a : A$	proof
$empty, unit$	$\bot, \top$
$A \oplus B$	$A \vee B$
$A \otimes B$	$A \wedge B$
$A \to B$	$A \Rightarrow B$
$A \to empty$	$\neg A$
$\sum x : A, B(x)$	$\exists x : A, B(x)$
$\prod x : A, B(x)$	$\forall x : A, B(x)$
$a = b$	equality $=$

The Curry-Howard-Voevodsky Correspondence[4]

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{(x,x) \mid x \in A\}$	path space A^I
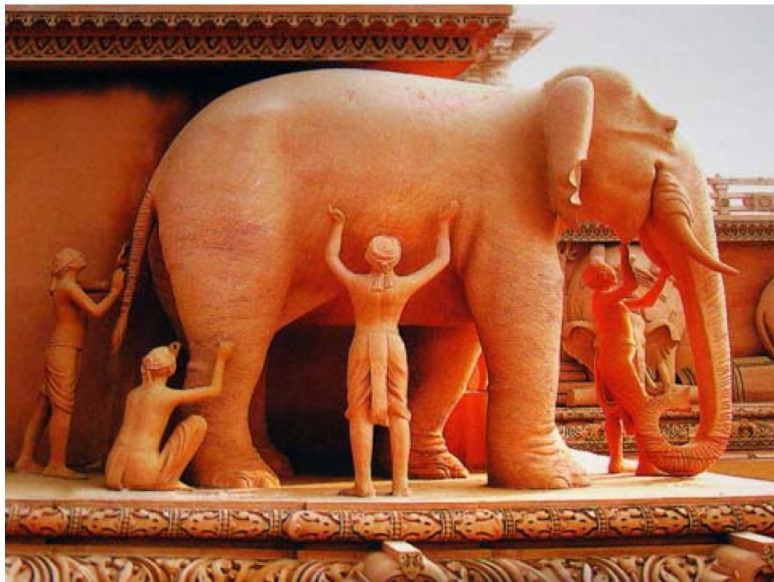
The Elephant

Let's Prove Some Theorems!

- Modus Ponens: $a \Rightarrow (a \Rightarrow b) \Rightarrow b$
- Modus Tollens: $\neg b \Rightarrow (a \Rightarrow b) \Rightarrow \neg a$
- Ex Falso Quodlibet: $\bot \Rightarrow a$
- Eagle: $(a \Rightarrow d \Rightarrow c) \Rightarrow a \Rightarrow (b \Rightarrow e \Rightarrow d) \Rightarrow b \Rightarrow e \Rightarrow c$
- Batman: $\neg\neg\neg a \Rightarrow \neg a$

Let's Prove Some Theorems!

- ▶ Modus Ponens: $a \Rightarrow (a \Rightarrow b) \Rightarrow b$
- ▶ Modus Tollens: $\neg b \Rightarrow (a \Rightarrow b) \Rightarrow \neg a$
- ▶ Ex Falso Quodlibet: $\perp \Rightarrow a$
- ▶ Eagle: $(a \Rightarrow d \Rightarrow c) \Rightarrow a \Rightarrow (b \Rightarrow e \Rightarrow d) \Rightarrow b \Rightarrow e \Rightarrow c$
- ▶ Batman: $\neg\neg\neg a \Rightarrow \neg a$
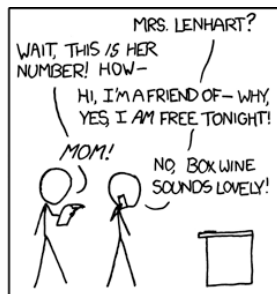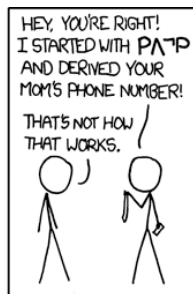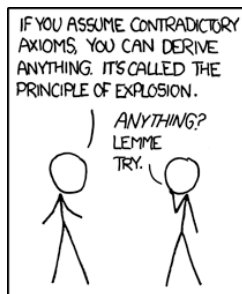
```
def modus_ponens : a → (a → b) → b := sorry

def modus_tollens : (b → empty) → (a → b) → (a → empty) := sorry

def ex_falso_quodlibet : empty → a := sorry

def eagle : (a → d → c) → a → (b → e → d) → b → e → c := sorry

def batman : (((a → empty) → empty) → empty) → (a → empty) := sorry
```

Principle of Explosion

Why total functions?

- Haskell, being a practical programming language, allows non-total functions and because of that is logically inconsistent

- To see why consider the function

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

Why total functions?

▶ Haskell, being a practical programming language, allows non-total functions and because of that is logically inconsistent

▶ To see why consider the function

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

▶ Lean disallows such definitions and even just assuming that such function exists yields a contradiction

```
variable fix {a : Type} : (a → a) → a

def wtf : empty := sorry
```

▶ Can you see how?

Function extensionality

```
def funext {f g : a → b} (h : Π x, f x = g x) : f = g
```

Function extensionality

```
def funext {f g : a → b} (h : Π x, f x = g x) : f = g
```

▶ Let's use it to prove:

```
def foo : (λ x, 2 * x) = (λ x, x + x) := sorry
```

Isomorphisms

```
structure iso (a b : Type) :=
(f : a → b) (g : b → a) (gf : Π x, g (f x) = x) (fg : Π x, f (g x) = x)

def inv {a b} : iso a b → iso b a := sorry

def comp {a b c} : iso a b → iso b c → iso a c := sorry
```

Equivalences

```
def fiber {a b} (f : a → b) (y : b) := Σ x : a, f x = y

def iscontr (a : Type) := Σ x : a, Π y : a, x = y

structure eqv (a b : Type) :=
(f : a → b) (h : Π y : b, iscontr (fiber f y))
```

Kan Extensions

```
def ran (g h : Type → Type) (a : Type) := Π b, (a → g b) → h b

def lan (g h : Type → Type) (a : Type) := Σ b, (g b → a) × h b
```

Perfectly Balanced Trees

```
inductive F (g : Type → Type) : Type → Type 1
| F0 : Π {a}, a → F a
| F1 : Π {a}, F (g a) → F a

inductive G (a : Type) : Type
| G0 : a → G
| G1 : a → G → G

inductive G23 (a : Type) : Type
| G2 : a → a → G23
| G3 : a → a → a → G23
```

Perfectly Balanced Trees (continued)

```
def iter {a} (g : a → a) : ℕ → a → a
| 0 := id
| (n + 1) := iter n ∘ g

def S (g : Type → Type) (a : Type) := Σ n : ℕ, iter g n a

def sf_iso {g a} : iso (S g a) (F g a) := sorry
```

Haskell Core Language

```haskell
data Expr b
  = Var    Id
  | Lit    Literal
  | App    (Expr b) (Arg b)
  | Lam    b (Expr b)
  | Let    (Bind b) (Expr b)
  | Case   (Expr b) b Type [Alt b]
  | Tick   (Tickish Id) (Expr b)
  | Type   Type
  | Cast   (Expr b) Coercion
  | Coercion Coercion

data Type
  = TyVarTy    Var
  | LitTy      TyLit
  | AppTy      Type Type
  | ForAllTy   !TyCoVarBinder Type
  | FunTy      Type Type
  | TyConApp   TyCon [KindOrType]
  | CastTy     Type KindCoercion
  | CoercionTy Coercion
```

Lean Core Language

```
inductive expr
| var         : nat → expr
| sort        : level → expr
| const       : name → list level → expr
| mvar        : name → name → expr → expr
| local_const : name → name → binder_info → expr → expr
| app         : expr → expr → expr
| lam         : name → binder_info → expr → expr → expr
| pi          : name → binder_info → expr → expr → expr
| elet        : name → expr → expr → expr → expr
| macro       : macro_def → list expr → expr
```

Example of Practical Applications: Crypto Algorithms

- **HACL***[5] is a formally verified cryptographic library in F*
- *HACL* stands for High-Assurance Cryptographic Library
- The goal of this library is to develop verified C reference implementations for popular cryptographic primitives and to verify them for memory safety, functional correctness, and secret independence
- The resulting generated C code is used in Mozilla Firefox and Wireguard

---

[5]https://github.com/project-everest/hacl-star

"The Relation of Mathematics and Physics"[6]
by Richard Feynman

- "Why nature is mathematical is, again, a mystery."

- "As people who looked at this thing as physicists cannot convert this thing to to any other language you have. If you want to discuss nature, to learn about nature, to appreciate nature, it's necessary to find out the language that she speaks in. She offers her information in only one form."

- "To summarize, I would use the words of Jeans, who said that **"the Great Architect seems to be a mathematician"**. To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty, of nature. If you want to learn about nature, to appreciate nature, it is necessary to understand the language that she speaks in."

---

[6]http://www.cornell.edu/video/
richard-feynman-messenger-lecture-2-relation-mathematics-physics

Further reading

- ▶ "Theorem Proving in Lean"
  https://leanprover.github.io/theorem_proving_in_lean

- ▶ "An Introduction to Lean"
  https://leanprover.github.io/introduction_to_lean

- ▶ "Constructive Mathematics and Computer Programming" by
  Per Martin-Löf

- ▶ "Proofs and Types" by Jean-Yves Girard

- ▶ "Physics, Topology, Logic and Computation: A Rosetta Stone" by
  John Baez and Mike Stay

- ▶ "Analytic Combinatorics" by Philippe Flajolet and Robert Sedgewick

- ▶ "Homotopy Type Theory: Univalent Foundations of Mathematics"
  by The Univalent Foundations Program

- ▶ Code and slides from this talk:
  https://github.com/skaslev/proofs-talk

Questions?