The internals of Lean

Floris van Doorn

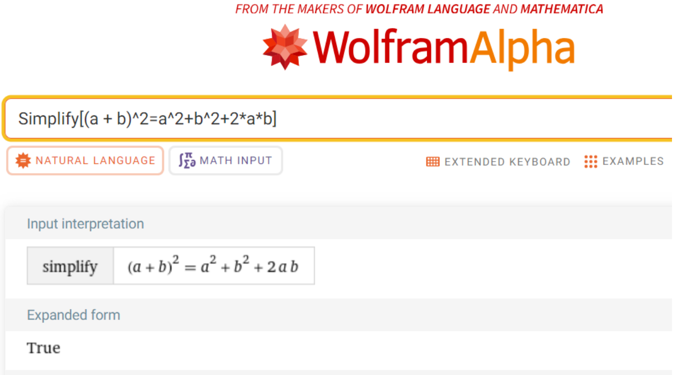
University of Bonn

25 January 2024

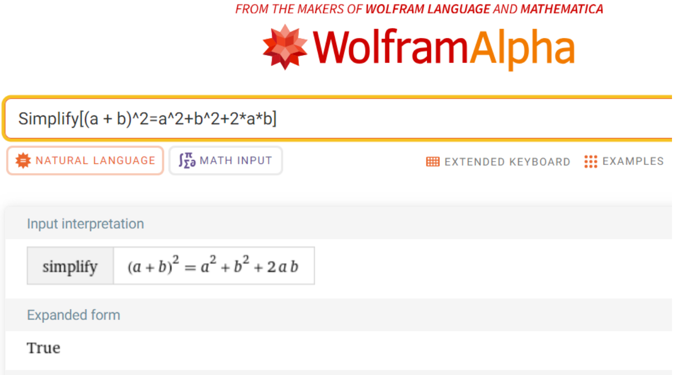
Questions discussed

- What is the logic of Lean?
- How does Lean check a proof?
- Can we trust proofs checked by Lean?

Computer algebra systems and proof assistants

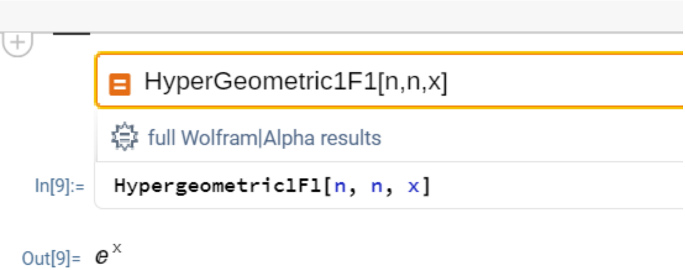


Computer algebra systems and proof assistants

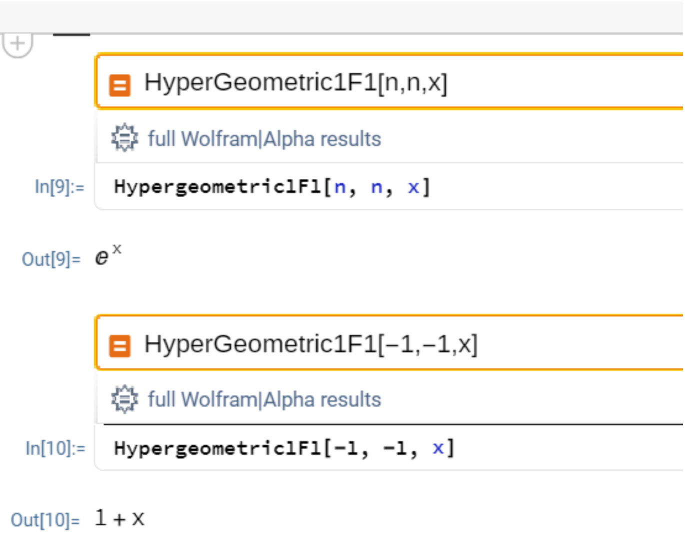



```
example (a b : ℂ) :
  (a + b)^2 = a^2 + b^2 + 2*a*b := by ring
```

Computer algebra systems and proof assistants



Computer algebra systems and proof assistants



Computer algebra systems and proof assistants

Computer Algebra System Perform efficient computations, that are correct most of the time.

Proof Assistant User writes a statement and proof, the program will check it.

Computer algebra systems and proof assistants

Computer Algebra System Perform efficient computations, that are correct most of the time.

Proof Assistant User writes a statement and proof, the program will check it.

Automated Theorem Prover User writes a statement, the program will find a proof or fail.

Logic of a proof assistant

A proof assistant implements a particular logic in which the proofs are checked.

Set theory Mizar, Metamath*

Simple type theory HOL Light, Isabelle*

Dependent type theory Lean, Coq

Logic of a proof assistant

A proof assistant implements a particular logic in which the proofs are checked.

- **Set theory** Mizar, Metamath*
- **Simple type theory** HOL Light, Isabelle*
- **Dependent type theory** Lean, Coq

You don't need to know the logic to start doing mathematics with a proof assistant

Objective of a proof assistant

- Check mathematical statements and definitions
- Check proofs
- Help with the proof

Objective of a proof assistant

- Check mathematical statements and definitions
- Check proofs
- Help with the proof

Mathematical statements often hide information.

We want to use + mean different things in different situations.

- $\pi + e$ is addition in $\mathbb{R}$
- $a + b$ (for a, b in some ring R) means addition in R
- $\aleph_1 + \aleph_3$ means addition of cardinal numbers

Objective of a proof assistant

- Check mathematical statements and definitions
- Check proofs
- Help with the proof

Mathematical statements often hide information.

We want to use + mean different things in different situations.

- $\pi + e$ is addition in $\mathbb{R}$
- $a + b$ (for a, b in some ring R) means addition in R
- $\aleph_1 + \aleph_3$ means addition of cardinal numbers

More complicated expressions:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

Type theory

In type theory, every term has an associated unique type.

$$3 : \mathbb{N}$$
$$\pi : \mathbb{R}$$
$$i : \mathbb{C}$$
$$\sin : \mathbb{R} \to \mathbb{R}$$

Type theory

In type theory, every term has an associated unique type.

$$3 : \mathbb{N}$$
$$\pi : \mathbb{R}$$
$$i : \mathbb{C}$$
$$\sin : \mathbb{R} \to \mathbb{R}$$

Type theory allows you to catch mistakes. If $f : \mathbb{R} \to \mathbb{R}$ then writing $f(i)$ will give a type error.

It will reject a statement like $3 \in \pi$ as nonsensical.

It can figure out the meaning of + depending on the type of the arguments.

Type theory

In type theory, every term has an associated unique type.

$$3 : \mathbb{N}$$
$$\pi : \mathbb{R}$$
$$i : \mathbb{C}$$
$$\sin : \mathbb{R} \to \mathbb{R}$$

Type theory allows you to catch mistakes. If $f : \mathbb{R} \to \mathbb{R}$ then writing $f(i)$ will give a type error.
It will reject a statement like $3 \in \pi$ as nonsensical.
It can figure out the meaning of + depending on the type of the arguments.

In set theory this is harder, it's "too flexible".

Warning

We have $3 : \mathbb{N}$ and $3 : \mathbb{R}$.

In type theory, these two 3's are not the same object.

(Of course, canonical inclusion $\mathbb{N} \hookrightarrow \mathbb{R}$ sends the former to the latter.)

In Lean, you can write `(3 : ℕ)` or `(3 : ℝ)` to force an expression to have a particular type.

Dependent type theory

Operations on types $\mathbb{Z} \times \mathbb{Q}$

Types can depend on values $\mathbb{R}^n$

Dependent type theory

Operations on types $\mathbb{Z} \times \mathbb{Q}$

Types can depend on values $\mathbb{R}^n$

Type universes `Type`

Propositions `Prop`

Dependent type theory

Operations on types $\mathbb{Z} \times \mathbb{Q}$

Types can depend on values $\mathbb{R}^n$

Type universes `Type`

Propositions `Prop`

(Dependent) Functions $n \mapsto \underbrace{(1, \frac{1}{2} \dots, \frac{1}{n})}_{\text{length } n} : (n : \mathbb{N}) \to \mathbb{R}^n$

Dependent type theory

Operations on types $\mathbb{Z} \times \mathbb{Q}$

Types can depend on values $\mathbb{R}^n$

Type universes `Type`

Propositions `Prop`

(Dependent) Functions $n \mapsto \underbrace{(1, \frac{1}{2} \ldots, \frac{1}{n})}_{\text{length } n} : (n : \mathbb{N}) \to \mathbb{R}^n$

Inductive types
```
inductive ℕ where
| zero : ℕ
| succ : ℕ → ℕ
```

Dependent type theory

Operations on types $\mathbb{Z} \times \mathbb{Q}$

Types can depend on values $\mathbb{R}^n$

Type universes `Type`

Propositions `Prop`

(Dependent) Functions $n \mapsto \underbrace{(1, \frac{1}{2} \dots, \frac{1}{n})}_{\text{length } n} : (n : \mathbb{N}) \to \mathbb{R}^n$

Inductive types
```
inductive ℕ where
  | zero : ℕ
  | succ : ℕ → ℕ
```

Definitional equality There is a notion of computation: $2 + 2 \equiv 4$, $(x, y).1 \equiv x$.

`rfl` can prove $a = b$ precisely when a and b are definitionally equal.

Dependent type theory

There are some details in Lean's type theory that are a bit complicated:

Useful to learn let-expressions, quotients, axiom of choice

A bit obscure universe levels, proof irrelevance, propositional extensionality

Very obscure impredicative `Prop`, subsingleton elimination, $\alpha\beta\delta\eta\zeta\iota$-conversion

Soundness

Is Lean's logic sound?

Short answer: Yes

Soundness

Is Lean's logic sound?

Yes, modulo issues with Gödels incompleteness theorem

Soundness

Is Lean's logic sound?

It is weaker than ZFC $+ \omega$ inaccessible cardinals

Lean

```
@[simp]
theorem integral_sin : ∫ x in a..b, sin x = cos a - cos b := by
  rw [integral_deriv_eq_sub' fun x => -cos x]
  · ring
  · norm_num
  · simp only [differentiableAt_neg_iff, differentiableAt_cos, implies_true]
  · exact continuousOn_sin
```

Processing a proof

What happens after writing a proof?

- Parsing (interpreting notation)
- Elaboration (figure out implicit information)
- Tactic execution
- Kernel checking

Elaboration

```
theorem add_comm {G : Type*} [AddCommMagma G]
  (a : G) (b : G) :
  a + b = b + a

example (a b c : ℝ) : a * b + c = c + a * b := by
  exact add_comm (a * b) c
```

- Lean figures out that `(G := ℝ)` from context (by looking at the type of `a`, `b` and `c`)
- Lean has a database of types where addition commutes, and looks up to see that it is true for $\mathbb{R}$ (*type-class inference*)

Tactic execution

Tactics can be any program that construct part of the proof.

Simple tactics that do 1 step in a proof: `intro`, `apply`, `have`, `rw`;
Domain-specific automation: `ring`, `linarith`
General automation: `simp`, `aesop`

Tactic execution

Tactics can be any program that construct part of the proof.

Simple tactics that do 1 step in a proof: `intro`, `apply`, `have`, `rw`;
Domain-specific automation: `ring`, `linarith`
General automation: `simp`, `aesop`

Running a tactics can result in

- Success: a finished proof
- Progess: 1 or more new goals to prove
- Failure: Raise an error

Tactic execution

Tactics can be any program that construct part of the proof.

Simple tactics that do 1 step in a proof: `intro`, `apply`, `have`, `rw`;
Domain-specific automation: `ring`, `linarith`
General automation: `simp`, `aesop`

Running a tactics can result in

- Success: a finished proof
- Progess: 1 or more new goals to prove
- Failure: Raise an error

Tactics produce a proof term.
(usually giant, unreadable for humans)

Kernel checking

- The kernel takes a proof term;
- Computes the type of this proof term;
- Checks that the type is the same as the theorem statement.

Kernel checking

- The kernel takes a proof term;
- Computes the type of this proof term;
- Checks that the type is the same as the theorem statement.

The kernel is a (relatively) small part of Lean, and it is the trusted codebase.

To trust that Lean only accepts true theorems, you only have to trust the kernel. You do not have to trust tactics.

Trust

To verify a formalization of non-malicious user:

- check the theorem statement
- check the definitions used in the statement
- check that Lean accepts the proof
- check that the authors didn't add axioms (`#print axioms my_theorem`)

Trust

To verify a formalization of non-malicious user:

- check the theorem statement
- check the definitions used in the statement
- check that Lean accepts the proof
- check that the authors didn't add axioms (`#print axioms my_theorem`)

If you are paranoid:

- check the proof with an external type checker
- check that the formalizers have not changed a notation or a definition
- verify the implementation of the external type checker

Trust

To verify a formalization of non-malicious user:

- check the theorem statement
- check the definitions used in the statement
- check that Lean accepts the proof
- check that the authors didn't add axioms (`#print axioms my_theorem`)

If you are paranoid:

- check the proof with an external type checker
- check that the formalizers have not changed a notation or a definition
- verify the implementation of the external type checker

If you are really paranoid:

- trust consistency of ZFC $+ \omega$ inaccessibles
- trust the compiler that compiled the type checker down to machine code
- trust that your hardware follows specifications
- trust that no cosmic rays interfered with your hard drive

Extensibility

Demo You can declare your own notation

```
notation3 "∫ "(...)" in "a".."b",
  "r:60:(scoped f ⇒ intervalIntegral f a b volume) ⇒ r
```

Extensibility

Demo You can declare your own notation

```
notation3 "∫ "(...)" in "a".."b",
  "r:60:(scoped f ⇒ intervalIntegral f a b volume) ⇒ r
```

You can declare your own tactics:

```
elab "my_assumption" : tactic ⇒ do
let target ← getMainTarget
for ldecl in ← getLCtx do
  if ldecl.isImplementationDetail then continue
  if ← isDefEq ldecl.type target then
    closeMainGoal ldecl.toExpr
    return
throwTacticEx `my_assumption (← getMainGoal)
  m!"no matching hypothesis of type {indentExpr target}"
```

Extensibility

Demo You can declare your own notation

```
notation3 "∫ "(...)" in "a".."b",
  "r:60:(scoped f ⇒ intervalIntegral f a b volume) ⇒ r
```

You can declare your own tactics:

```
elab "my_assumption" : tactic ⇒ do
let target ← getMainTarget
for ldecl in ← getLCtx do
  if ldecl.isImplementationDetail then continue
  if ← isDefEq ldecl.type target then
    closeMainGoal ldecl.toExpr
    return
throwTacticEx `my_assumption (← getMainGoal)
  m!"no matching hypothesis of type {indentExpr target}"
```

You can even declare your language, and write a parser for that language.

Extensibility

Demo You can declare your own notation

```
notation3 "∫ "(...)" in "a".."b",
  "r:60:(scoped f ⇒ intervalIntegral f a b volume) ⇒ r
```

You can declare your own tactics:

```
elab "my_assumption" : tactic ⇒ do
let target ← getMainTarget
for ldecl in ← getLCtx do
  if ldecl.isImplementationDetail then continue
  if ← isDefEq ldecl.type target then
    closeMainGoal ldecl.toExpr
    return
throwTacticEx `my_assumption (← getMainGoal)
  m!"no matching hypothesis of type {indentExpr target}"
```

You can even declare your language, and write a parser for that language.

In fact, almost every part of Lean (parsing, elaboration, tactics, compilation) are written in Lean

Conclusions

- Type theory is a useful logic for formalization;
- You can trust Lean formalizations;
- Lean is very extensible.