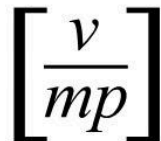




LUIS WIRTH

**Lean4 and the Curry-Howard
Isomorphism: The Deep Connection
Between Logic and Programming
Through Type Theory**



Tuesday 05/11
12:15-13:00
ETH HG G3



Why Lean matters



Why Lean matters

Lean = Programming Language + Interactive Proof Assistant

Why Lean matters

Lean = Programming Language + Interactive Proof Assistant

Allows us to do some groundbreaking things:

- *Express* Mathematical Ideas using code.
- *Express* Mathematical Proofs using code.
- *Verify* the Proofs formally.
- Write in a cool ass programming language

Why Lean matters

Lean = Programming Language + Interactive Proof Assistant

Allows us to do some groundbreaking things:

- *Express* Mathematical Ideas using code.
- *Express* Mathematical Proofs using code.
- *Verify* the Proofs formally.
- Write in a cool ass programming language

Lean = Type Theory + Lambda-Calculus

Why Lean matters

Lean = Programming Language + Interactive Proof Assistant

Allows us to do some groundbreaking things:

- *Express* Mathematical Ideas using code.
- *Express* Mathematical Proofs using code.
- *Verify* the Proofs formally.
- Write in a cool ass programming language

Lean = Type Theory + Lambda-Calculus

Type Theory $\rightsquigarrow$ Type System/Checker

Lambda-Calculus $\rightsquigarrow$ Functional Programming

Why Type Theory matters

Why Type Theory matters

New **Foundation for Mathematics** beyond traditional set theory.

“Set Theory is actually not so nice...” - Some Category Theorist probably

Why Type Theory matters

New **Foundation for Mathematics** beyond traditional set theory.

“Set Theory is actually not so nice...” - Some Category Theorist probably

Curry-Howard Isomorphism (CH) tells us that logic can be done in a type system!



—



Isomorphism.

Why Type Theory matters

New **Foundation for Mathematics** beyond traditional set theory.

“Set Theory is actually not so nice...” - Some Category Theorist probably

Curry-Howard Isomorphism (CH) tells us that logic can be done in a type system!

Propositions-as-Types
Proofs-as-Programs

Mathematicians use Lean

Lean enables **large-scale collaboration** by **breaking down complex proofs** into smaller parts.

Lean also **ensures the correctness of proofs**.

With Lean we begin to see **how AI can accelerate the formalization of mathematics**, opening up new possibilities for research.

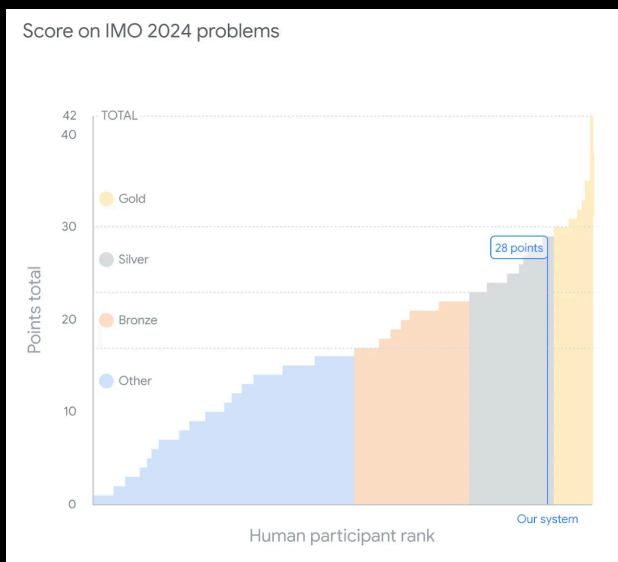
— Terence Tao, Fields Medalist and Professor of Mathematics, UCLA

AI Breakthrough in Math, July 2024

“AI achieves silver-medal standard
solving International Mathematical Olympiad problems”

AI Breakthrough in Math, July 2024

“AI achieves silver-medal standard solving International Mathematical Olympiad problems”



Google DeepMind's AlphaProof

AI Breakthrough in Math, July 2024

“AI achieves silver-medal standard solving International Mathematical Olympiad problems”



Google DeepMind's AlphaProof

The fact that the program can come up with a non-obvious construction like this is very impressive, and well **beyond what I thought was state of the art.**

— Prof Sir Timothy Gowers, **IMO gold medalist** and **Fields Medal winner**

AI Breakthrough in Math, July 2024

“AI achieves silver-medal standard solving International Mathematical Olympiad problems”



Google DeepMind's AlphaProof

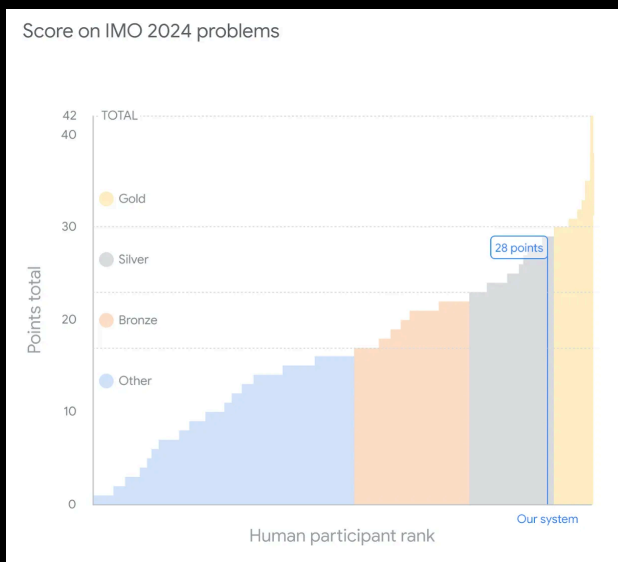
The fact that the program can come up with a non-obvious construction like this is very impressive, and well **beyond what I thought was state of the art.**

— Prof Sir Timothy Gowers, **IMO gold medalist** and **Fields Medal winner**

Soon (this or next year) Gold Medal probably.

AI Breakthrough in Math, July 2024

“AI achieves silver-medal standard solving International Mathematical Olympiad problems”



Google DeepMind's AlphaProof

The fact that the program can come up with a non-obvious construction like this is very impressive, and well **beyond what I thought was state of the art.**

— Prof Sir Timothy Gowers, **IMO gold medalist** and **Fields Medal winner**

Soon (this or next year) Gold Medal probably.

Maybe someday Millenium Problem?

The Typing Judgment

$$t : T$$

“ t is a term of type T ”

Lean is just a Programming Language!

Lean is just a Programming Language!

Let's define some objects.

Lean is just a Programming Language!

Let's define some objects: constants!

Specify **name**, **type** and **term**.

```
def n : Nat    := 69
def z : Int     := -420
def f : Float   := 8.0085
def b : Bool    := true
```

Lean is just a Programming Language!

Let's define some objects: constants!

Specify **name**, **type** (optional) and **term**.

```
def n      := 69
def z      := -420
def f      := 8.0085
def b      := true
```

Lean is just a Programming Language!

Let's define some objects: constants!

Specify **name**, **type** (optional) and **term**.

```
def n      := 69
def z      := -420
def f      := 8.0085
def b      := true
```

Use **#check** command to deduce type of expression.

```
#check n      -- Nat
#check z      -- Int
#check 5 * (n + 0) -- Nat
```

Lean is just a Programming Language!

Let's define some objects: constants!

Specify **name**, **type** (optional) and **term**.

```
def n      := 69
def z      := -420
def f      := 8.0085
def b      := true
```

Use **#check** command to deduce type of expression.

```
#check n      -- Nat
#check z      -- Int
#check 5 * (n + 0) -- Nat
```

Use **#eval** command to evaluate expression to value.

```
#eval 5 * 4 -- 20
#eval n + 2 -- 71
```

Type Constructors

Build new types from existing types.

Type Constructors

Build new types from existing types.

Product Types

Type Constructors

Build new types from existing types.

Product Types

```
def p : Int × Bool := (2, true)
#check p -- Int × Bool
```

Type Constructors

Build new types from existing types.

Product Types

```
def p : Int × Bool := (2, true)
#check p -- Int × Bool
```

The two terms exist side by side.

Type Constructors

Build new types from existing types.

Product Types

```
def p : Int × Bool := (2, true)
#check p -- Int × Bool
```

The two terms exist side by side.

The C++ equivalent:

Type Constructors

Build new types from existing types.

Product Types

```
def p : Int × Bool := (2, true)
#check p -- Int × Bool
```

The two terms exist side by side.

The C++ equivalent:

A struct with two fields.

```
struct Prod {
    int l;
    bool r;
} p = {2, true};
```

Type Constructors

Another one, built different.

Sum Types

Type Constructors

Another one, built different.

Sum Types

```
def s0 : Int ⊕ Bool := Sum.inl 5
def s1 : Int ⊕ Bool := Sum.inr true
#check s1 -- Int ⊕ Bool
```

Type Constructors

Another one, built different.

Sum Types

```
def s0 : Int ⊕ Bool := Sum.inl 5
def s1 : Int ⊕ Bool := Sum.inr true
#check s1 -- Int ⊕ Bool
```

It's either the left or the right term.

Type Constructors

Another one, built different.

Sum Types

```
def s0 : Int ⊕ Bool := Sum.inl 5
def s1 : Int ⊕ Bool := Sum.inr true
#check s1 -- Int ⊕ Bool
```

It's either the left or the right term.

The C++ equivalent:

Type Constructors

Another one, built different.

Sum Types

```
def s0 : Int ⊕ Bool := Sum.inl 5
def s1 : Int ⊕ Bool := Sum.inr true
#check s1 -- Int ⊕ Bool
```

It's either the left or the right term.

The C++ equivalent:

A union with two variants.

```
union Sum {
    int l;
    bool r;
};
Sum s0 = {5};
Sum s1 = {true};
```

Type Constructors

Another one, built different.

Sum Types

```
def s0 : Int ⊕ Bool := Sum.inl 5
def s1 : Int ⊕ Bool := Sum.inr true
#check s1 -- Int ⊕ Bool
```

It's either the left or the right term.

The C++ equivalent:

A *tagged* union with two variants.

```
struct Sum {
    enum Variant { Left, Right } variant;
    union { int l; bool r; } data;

    Sum(int l) : data{l}, variant(Left) {}
    Sum(bool r) : data{r}, variant(Right) {}
};
Sum s0 = {5};
Sum s1 = {true};
```

Functions in Lean

Functions in Lean

Define a simple function (**Lambda Abstraction**):

Functions in Lean

Define a simple function (**Lambda Abstraction**):

```
def plus_one (x : Nat) : Nat := x + 1
```

Looks like argument-dependent constant.

Functions in Lean

Define a simple function (**Lambda Abstraction**):

```
def plus_one (x : Nat) : Nat := x + 1
```

Looks like argument-dependent constant.

Apply the function to an argument (**Lambda Application**):

Functions in Lean

Define a simple function (**Lambda Abstraction**):

```
def plus_one (x : Nat) : Nat := x + 1
```

Looks like argument-dependent constant.

Apply the function to an argument (**Lambda Application**):

```
#eval plus_one 5 -- 6
```

No paranthesis needed.

Lean is a *functional* programming language!

Functions are first class objects.

Consider our function:

```
def plus_one (x : Nat) : Nat := x + 1
```

Lean is a *functional* programming language!

Functions are first class objects.

Consider our function:

```
def plus_one (x : Nat) : Nat := x + 1
```

It's an object! What is it's type?

Lean is a *functional* programming language!

Functions are first class objects.

Consider our function:

```
def plus_one (x : Nat) : Nat := x + 1
```

It's an object! What is it's type?

```
#check plus_one
```

Lean is a *functional* programming language!

Functions are first class objects.

Consider our function:

```
def plus_one (x : Nat) : Nat := x + 1
```

It's an object! What is it's type?

```
#check plus_one -- Nat → Nat
```

Lean is a *functional* programming language!

Functions are first class objects.

Consider our function:

```
def plus_one (x : Nat) : Nat := x + 1
```

It's an object! What is it's type?

```
#check plus_one -- Nat → Nat
```

New type constructor just dropped: →

Lean is a *functional* programming language!

Functions are first class objects.

Consider our function:

```
def plus_one (x : Nat) : Nat := x + 1
```

It's an object! What is it's type?

```
#check plus_one -- Nat → Nat
```

New type constructor just dropped: $\rightarrow$

Given two types A and B, we get the new type $A \rightarrow B$, consisting of all functions from A to B.

Different way to define a function:

Lean is a *functional* programming language!

Functions are first class objects.

Consider our function:

```
def plus_one (x : Nat) : Nat := x + 1
```

It's an object! What is it's type?

```
#check plus_one -- Nat → Nat
```

New type constructor just dropped: $\rightarrow$

Given two types A and B, we get the new type $A \rightarrow B$, consisting of all functions from A to B.

Different way to define a function:

```
def plus_one : Nat → Nat := λ x ↦ x + 1  
#check (λ x ↦ x + 1) -- Nat → Nat
```

Directly using a lambda term of type $\text{Nat} \rightarrow \text{Nat}$.

Higher-Order functions

Functions with another function as argument.

Higher-Order functions

Functions with another function as argument.

We can write this function for instance:

```
def do_twice (f : Nat → Nat) (x : Nat) : Nat :=  
  f (f x)
```

Higher-Order functions

Functions with another function as argument.

We can write this function for instance:

```
def do_twice (f : Nat → Nat) (x : Nat) : Nat :=  
  f (f x)  
#eval do_twice plus_one 5 -- 7
```

Higher-Order functions

Functions with another function as argument.

We can write this function for instance:

```
def do_twice (f : Nat → Nat) (x : Nat) : Nat :=  
  f (f x)  
#eval do_twice plus_one 5 -- 7
```

In C++:

```
int do_twice(std::function<int(int)> f, int x) {  
  return f(f(x));  
}
```

Multiple Arguments

Functional Programmers like Curry 😊.

Multiple Arguments

Functional Programmers like Curry 🥰.

In non-functional languages:

Multiple Arguments

Functional Programmers like Curry 😊.

In non-functional languages: With tuples!

```
def sum_tuple : Nat × Nat → Nat := λ (x, y) ↦ x + y
```

Multiple Arguments

Functional Programmers like Curry 😊.

In non-functional languages: With tuples!

```
def sum_tuple : Nat × Nat → Nat := λ (x, y) ↦ x + y
```

Instead in functional languages: Function Chain (**Currying**).

```
def sum_curry : Nat → (Nat → Nat) := λ x ↦ λ y ↦ x + y
#check sum_curry      -- Nat -> Nat -> Nat
```

Multiple Arguments

Functional Programmers like Curry 😊.

In non-functional languages: With tuples!

```
def sum_tuple : Nat × Nat → Nat := λ (x, y) ↦ x + y
```

Instead in functional languages: Function Chain (**Currying**).

```
def sum_curry : Nat → (Nat → Nat) := λ x ↦ λ y ↦ x + y
#check sum_curry      -- Nat -> Nat -> Nat
#check sum_curry 5     -- Nat -> Nat
```

Multiple Arguments

Functional Programmers like Curry 😊.

In non-functional languages: With tuples!

```
def sum_tuple : Nat × Nat → Nat := λ (x, y) ↦ x + y
```

Instead in functional languages: Function Chain (**Currying**).

```
def sum_curry : Nat → (Nat → Nat) := λ x ↦ λ y ↦ x + y
#check sum_curry      -- Nat -> Nat -> Nat
#check sum_curry 5    -- Nat -> Nat
#check sum_curry 5 6  -- Nat
```

Multiple Arguments

Functional Programmers like Curry 😊.

In non-functional languages: With tuples!

```
def sum_tuple : Nat × Nat → Nat := λ (x, y) ↦ x + y
```

Instead in functional languages: Function Chain (**Currying**).

```
def sum_curry : Nat → (Nat → Nat) := λ x ↦ λ y ↦ x + y
#check sum_curry      -- Nat -> Nat -> Nat
#check sum_curry 5     -- Nat -> Nat
#check sum_curry 5 6   -- Nat
#eval  sum_curry 5 6   -- 11
```

Multiple Arguments

Functional Programmers like Curry 😊.

In non-functional languages: With tuples!

```
def sum_tuple : Nat × Nat → Nat := λ (x, y) ↦ x + y
```

Instead in functional languages: Function Chain (**Currying**).

```
def sum_curry : Nat → (Nat → Nat) := λ x ↦ λ y ↦ x + y
#check sum_curry      -- Nat -> Nat -> Nat
#check sum_curry 5     -- Nat -> Nat
#check sum_curry 5 6    -- Nat
#eval  sum_curry 5 6    -- 11
```

Allows for **partial function application**.

Types as Objects!

We can declare new constants for types.

Types as Objects!

We can declare new constants for types.

```
def  $\alpha$  := Nat  
def  $\beta$  := Bool
```

Types as Objects!

We can declare new constants for types.

```
def  $\alpha$  := Nat  
def  $\beta$  := Bool
```

We can use it to specify the type of a “data” constant.

```
def n :  $\alpha$  := 5
```

Types as Objects!

We can declare new constants for types.

```
def  $\alpha$  := Nat  
def  $\beta$  := Bool
```

We can use it to specify the type of a “data” constant.

```
def n :  $\alpha$  := 5
```

Actually not so crazy. In C++ one can also define type constants.

Types as Objects!

We can declare new constants for types.

```
def  $\alpha$  := Nat  
def  $\beta$  := Bool
```

We can use it to specify the type of a “data” constant.

```
def n :  $\alpha$  := 5
```

Actually not so crazy. In C++ one can also define type constants.

```
using T = int;  
T n;
```

Types as Objects!

We can declare new constants for types.

```
def  $\alpha$  := Nat  
def  $\beta$  := Bool
```

We can use it to specify the type of a “data” constant.

```
def n :  $\alpha$  := 5
```

Actually not so crazy. In C++ one can also define type constants.

```
using T = int;  
T n;
```

But this is not really an object. Not as flexible.

In Lean this is like any other object.

But every object must have a type...

Given that this is an object.

```
def  $\alpha$  := Nat
```

But every object must have a type...

Given that this is an object.

```
def  $\alpha$  := Nat
```

What is it's type?

```
#check  $\alpha$ 
```

But every object must have a type...

Given that this is an object.

```
def  $\alpha$  := Nat
```

What is it's type?

```
#check  $\alpha$  -- Type
```

But every object must have a type...

Given that this is an object.

```
def  $\alpha$  := Nat
```

What is it's type?

```
#check  $\alpha$  -- Type
```

So we have now discovered a type that is called `Type`.

We can write fill in our definition as:

```
def  $\alpha$  : Type := Nat
```

Generic Functions using **Type**

We can take a **Type** argument to make a function generic.

Generic Functions using **Type**

We can take a **Type** argument to make a function generic.

```
def do_twice (f : Nat → Nat) (x : Nat) : Nat :=  
  f (f x)
```

Generic Functions using **Type**

We can take a **Type** argument to make a function generic.

```
def do_twice ( $\alpha$  : Type) (f :  $\alpha \rightarrow \alpha$ ) (x :  $\alpha$ ) :  $\alpha$  :=  
  f (f x)
```

Generic Functions using **Type**

We can take a **Type** argument to make a function generic.

```
def do_twice ( $\alpha$  : Type) (f :  $\alpha \rightarrow \alpha$ ) (x :  $\alpha$ ) :  $\alpha$  :=  
  f (f x)
```

In C++ we can do generics using templates:

```
template <typename T>  
T do_twice(std::function<T(T)> f, T x) {  
  return f(f(x));  
}
```

Generic Functions using **Type**

We can take a **Type** argument to make a function generic.

```
def do_twice ( $\alpha$  : Type) (f :  $\alpha \rightarrow \alpha$ ) (x :  $\alpha$ ) :  $\alpha$  :=  
  f (f x)
```

In C++ we can do generics using templates:

```
template <typename T>  
T do_twice(std::function<T(T)> f, T x) {  
  return f(f(x));  
}
```

But here C++ makes a big distinction between type variables and normal variables.
Lean doesn't. It's like any other object.

Type Constructors on **Type**

Since **Type** is a type, we can use the type constructors we have seen so far on it.

Type Constructors on **Type**

Since **Type** is a type, we can use the type constructors we have seen so far on it.

Function arrow type constructor can be used.

A function on types.

```
def f : Type → Type := λ _ => Nat
```

Type Constructors on **Type**

Since **Type** is a type, we can use the type constructors we have seen so far on it.

Function arrow type constructor can be used.

A function on types.

```
def f : Type → Type := λ _ => Nat
```

Type Constructors on **Type**

Since **Type** is a type, we can use the type constructors we have seen so far on it.

Function arrow type constructor can be used.

A function on types.

```
def f : Type → Type := λ _ => Nat
```

Now we can understand what Prod is.

```
#check Int × Bool      -- Type
#check Prod Int Bool    -- Type
#check Prod Int         -- Type -> Type
#check Prod             -- Type -> Type -> Type
```

Note:

Actually Prod is even more general, and could also take **Type** as argument.

We could have Prod **Type Type**. It abstracts over “type universes”.

Type Constructors on **Type**

Since **Type** is a type, we can use the type constructors we have seen so far on it.

Function arrow type constructor can be used.

A function on types.

```
def f : Type → Type := λ _ => Nat
```

Now we can understand what Prod is.

```
#check Int × Bool      -- Type
#check Prod Int Bool   -- Type
#check Prod Int        -- Type -> Type
#check Prod             -- Type -> Type -> Type
```

Note:

Actually Prod is even more general, and could also take **Type** as argument.

We could have Prod **Type Type**. It abstracts over “type universes”.

In C++ using generics.

```
template <typename L, typename R>
struct Prod {
    L l;
    R r;
}
```

```
Prod<int, bool> p;
```

What's the Type of **Type**?

Type must be an object too, since it's a type.

What's the Type of **Type**?

Type must be an object too, since it's a type.

We must be able to define such a constant.

```
def t := Type
```

What's the Type of **Type**?

Type must be an object too, since it's a type.

We must be able to define such a constant.

```
def t := Type
```

What is the type of this constant?

```
#check t
```

What's the Type of **Type**?

Type must be an object too, since it's a type.

We must be able to define such a constant.

```
def t := Type
```

What is the type of this constant?

```
#check t -- Type 1
```

What's the Type of **Type**?

Type must be an object too, since it's a type.

We must be able to define such a constant.

```
def t := Type
```

What is the type of this constant?

```
#check t -- Type 1
```

So there's a new type, called **Type 1**. It once again must be an object.

What's the Type of **Type**?

Type must be an object too, since it's a type.

We must be able to define such a constant.

```
def t := Type
```

What is the type of this constant?

```
#check t -- Type 1
```

So there's a new type, called **Type 1**. It once again must be an object.

```
def t1 := Type 1
```

```
#check t1 -- Type 2
```

What's the Type of **Type**?

Type must be an object too, since it's a type.

We must be able to define such a constant.

```
def t := Type
```

What is the type of this constant?

```
#check t -- Type 1
```

So there's a new type, called **Type 1**. It once again must be an object.

```
def t1 := Type 1
#check t1 -- Type 2
```

Keep on doing this:

```
#check Nat      -- Type
#check Type     -- Type 1
#check Type 1   -- Type 2
#check Type 2   -- Type 3
#check Type 3   -- Type 4
-- ...
```

We've stumbled upon the infinitely many, but countable **type universes**.

Hierarchy of Type Universes

Each lower type universe is a term in it's next higher type universe.

```
def t0 : Type 0 := Nat
def t1 : Type 1 := Type 0
def t2 : Type 2 := Type 1
def t3 : Type 3 := Type 2
-- ...
```

In general `Type n : Type (n+1)`.

Hierarchy of Type Universes

Each lower type universe is a term in it's next higher type universe.

```
def t0 : Type 0 := Nat
def t1 : Type 1 := Type 0
def t2 : Type 2 := Type 1
def t3 : Type 3 := Type 2
-- ...
```

In general `Type n : Type (n+1)`.

The distinction between universe, type and term becomes blurry.
It depends on the context.

Universe	Type 0	Type 1	Type 2	Type 3	...
Type	Nat	Type 0	Type 1	Type 2	...
Term	1	Nat	Type 0	Type 1	...

What's the use of Higher Order Types?

It's all about abstraction.

What's the use of Higher Order Types?

It's all about abstraction.

Standard Type Universe `Type 0`. Who lives here?

- Data types: `Nat : Type`, `String : Type`
- Function Types on Data `Nat → Nat : Type`.

What's the use of Higher Order Types?

It's all about abstraction.

Standard Type Universe `Type 0`. Who lives here?

- Data types: `Nat : Type`, `String : Type`
- Function Types on Data `Nat → Nat : Type`.

One abstraction level higher: `Type 1`. Who lives here?

- `Type 0` itself: `Type 0 : Type 1`.
- Type constructors: `Type → Type → Type : Type 1`

What's the use of Higher Order Types?

It's all about abstraction.

Standard Type Universe **Type 0**. Who lives here?

- Data types: $\text{Nat} : \text{Type}$, $\text{String} : \text{Type}$
- Function Types on Data $\text{Nat} \rightarrow \text{Nat} : \text{Type}$.

One abstraction level higher: **Type 1**. Who lives here?

- **Type 0** itself: $\text{Type } 0 : \text{Type } 1$.
- Type constructors: $\text{Type} \rightarrow \text{Type} \rightarrow \text{Type} : \text{Type } 1$

Abstracting over all type constructors? **Type 2**!

- Functions from type constructor to other type constructor.

What's the use of Higher Order Types?

It's all about abstraction.

Standard Type Universe **Type 0**. Who lives here?

- Data types: $\text{Nat} : \text{Type}$, $\text{String} : \text{Type}$
- Function Types on Data $\text{Nat} \rightarrow \text{Nat} : \text{Type}$.

One abstraction level higher: **Type 1**. Who lives here?

- **Type 0** itself: $\text{Type 0} : \text{Type 1}$.
- Type constructors: $\text{Type} \rightarrow \text{Type} \rightarrow \text{Type} : \text{Type 1}$

Abstracting over all type constructors? **Type 2**!

- Functions from type constructor to other type constructor.

Arbitrary levels of abstraction are possible!

Type n

Does the typed Barber shave itself?

Why not just have one universe.

One type to rule them all: The type of ALL types.

Does the typed Barber shave itself?

Why not just have one universe.

One type to rule them all: The type of ALL types.

Oh oh. Russell is knocking on the door. And he has a Paradox for you.

Does the typed Barber shave itself?

Why not just have one universe.

One type to rule them all: The type of ALL types.

Oh oh. Russell is knocking on the door. And he has a Paradox for you.

$$R = \{x \mid x \notin x\} \text{ then } R \in R \iff R \notin R \text{ ⚡}$$

Does the typed Barber shave itself?

Why not just have one universe.

One type to rule them all: The type of ALL types.

Oh oh. Russell is knocking on the door. And he has a Paradox for you.

$$R = \{x \mid x \notin x\} \text{ then } R \in R \iff R \notin R \text{ ⚡}$$

Russell invented simple type theory!

Does the typed Barber shave itself?

Why not just have one universe.

One type to rule them all: The type of ALL types.

Oh oh. Russell is knocking on the door. And he has a Paradox for you.

$$R = \{x \mid x \notin x\} \text{ then } R \in R \iff R \notin R \text{ ⚡}$$

Russell invented simple type theory!

Girard's Paradox is type-theoretic version of it.

Solution: *Strict* hierarchy of type universes.

No universe may contain itself.

```
def t : Type 0 := Type 0 -- ERROR: type mismatch: `Type 0` actually has type `Type 1`
```

How to do Math in Lean?

Spill the tea!

How to do Math in Lean?

Spill the tea!

Logical Interpretation of Type Theory .

Leverage the Type System!

~> **Mathematical Statements need to be types!**

How to do Math in Lean?

Spill the tea!

Logical Interpretation of Type Theory .

Leverage the Type System!

~> **Mathematical Statements need to be types!**

Start by expressing **Propositional Logic** using types.

Conjunction	$p \wedge q$
Disjunction	$p \vee q$
Implication	$p \rightarrow q$
Negation	$\neg p$
True	$\top$
False	$\perp$

Propositions in Lean

New Type Universe just dropped: `Prop`

Universe of all logical propositions.

Propositions in Lean

New Type Universe just dropped: **Prop**

Universe of all logical propositions.

Types in **Prop are propositions.**

For example: $p : \text{Prop}$ and $q : \text{Prop}$.

Propositions in Lean

New Type Universe just dropped: **Prop**

Universe of all logical propositions.

Types in **Prop are propositions.**

For example: $p : \text{Prop}$ and $q : \text{Prop}$.

What's the meaning of terms $hp : p$ and $hq : q$?

Herein lies the magic! **A term is a proof!**

Propositions in Lean

New Type Universe just dropped: **Prop**

Universe of all logical propositions.

Types in **Prop are propositions.**

For example: $p : \text{Prop}$ and $q : \text{Prop}$.

What's the meaning of terms $hp : p$ and $hq : q$?

Herein lies the magic! **A term is a proof!**

$$t : T \iff t \text{ is a witness to the truth of } T$$

Propositions in Lean

New Type Universe just dropped: **Prop**

Universe of all logical propositions.

Types in **Prop are propositions.**

For example: $p : \text{Prop}$ and $q : \text{Prop}$.

What's the meaning of terms $hp : p$ and $hq : q$?

Herein lies the magic! **A term is a proof!**

$$t : T \iff t \text{ is a witness to the truth of } T$$

To prove a proposition is to construct its term!

Lean can validate the witness/proof by just typechecking it!

Propositions in Lean

New Type Universe just dropped: **Prop**

Universe of all logical propositions.

Types in **Prop are propositions.**

For example: $p : \text{Prop}$ and $q : \text{Prop}$.

What's the meaning of terms $hp : p$ and $hq : q$?

Herein lies the magic! **A term is a proof!**

$$t : T \iff t \text{ is a witness to the truth of } T$$

To prove a proposition is to construct its term!

Lean can validate the witness/proof by just typechecking it!

There could be multiple witnesses to the same proposition.

Propositions in Lean

New Type Universe just dropped: **Prop**

Universe of all logical propositions.

Types in **Prop are propositions.**

For example: $p : \text{Prop}$ and $q : \text{Prop}$.

What's the meaning of terms $hp : p$ and $hq : q$?

Herein lies the magic! **A term is a proof!**

$$t : T \iff t \text{ is a witness to the truth of } T$$

To prove a proposition is to construct its term!

Lean can validate the witness/proof by just typechecking it!

There could be multiple witnesses to the same proposition.

We don't care: **Proof-Irrelevance!** We only care about **Inhabitedness**.

T is inhabited $\Leftrightarrow$ there are terms of type T

T is uninhabited $\Leftrightarrow$ there are no terms of type T

Functions between Propositions

We have two arbitrary propositions

```
variable (p q : Prop)
```

What is the meaning of a function $f : p \rightarrow q$?

Functions between Propositions

We have two arbitrary propositions

```
variable (p q : Prop)
```

What is the meaning of a function $f : p \rightarrow q$?

Syntactically looks like implication...

Functions between Propositions

We have two arbitrary propositions

```
variable (p q : Prop)
```

What is the meaning of a function $f : p \rightarrow q$?

Syntactically looks like implication...

Also semantically the same! How?

Functions between Propositions

We have two arbitrary propositions

```
variable (p q : Prop)
```

What is the meaning of a function $f : p \rightarrow q$?

Syntactically looks like implication...

Also semantically the same! How?

Given proof $hp : p$ and function $f : p \rightarrow q$, we can get $hq : q := f\ hp$.

Known as **modus ponens** or **implication elimination**.

Functions between Propositions

We have two arbitrary propositions

```
variable (p q : Prop)
```

What is the meaning of a function $f : p \rightarrow q$?

Syntactically looks like implication...

Also semantically the same! How?

Given proof $hp : p$ and function $f : p \rightarrow q$, we can get $hq : q := f\ hp$.

Known as **modus ponens** or **implication elimination**.

How to do **implication introduction**?

Create a function $f : p \rightarrow q$ by assuming $hp : p$ and deriving $hq : q$.

Functions between Propositions

We have two arbitrary propositions

```
variable (p q : Prop)
```

What is the meaning of a function $f : p \rightarrow q$?

Syntactically looks like implication...

Also semantically the same! How?

Given proof $hp : p$ and function $f : p \rightarrow q$, we can get $hq : q := f\ hp$.

Known as **modus ponens** or **implication elimination**.

How to do **implication introduction**?

Create a function $f : p \rightarrow q$ by assuming $hp : p$ and deriving $hq : q$.

First instance of **Curry-Howard Isomorphism (CH)**!

Functions are Implications.

Logical Conjunction $p \wedge q$

What is the corresponding type?

Logical Conjunction $p \wedge q$

What is the corresponding type?

Type combining to terms of type p and q ?

Logical Conjunction $p \wedge q$

What is the corresponding type?

Type combining to terms of type p and q ?

Product Type $p \times q$ by CH!

Logical Conjunction $p \wedge q$

What is the corresponding type?

Type combining to terms of type p and q ?

Product Type $p \times q$ by CH!

Illustrate at example!

```
theorem and_commutative (p q : Prop) : p ∧ q → q ∧ p :=  
  λ hpq : p ∧ q ↦ And.intro hpq.right hpq.left
```

Logical Disjunction $p \vee q$

What is the corresponding type?

Logical Disjunction $p \vee q$

What is the corresponding type?

Sum Type $p \oplus q$ by CH!

Logical Disjunction $p \vee q$

What is the corresponding type?

Sum Type $p \oplus q$ by CH!

```
theorem or_commutative (p q : Prop) : p ∨ q → q ∨ p :=  
  λ hpq : p ∨ q ↦  
    hpq.elim  
      (λ hp ↦ Or.inr hp)  
      (λ hq ↦ Or.inl hq)
```

Logical Constant True

Logical Constant True

True is simply true.

Get Proof for free:

```
def t : True := True.intro
```

Logical Constant True

True is simply true.

Get Proof for free:

```
def t : True := True.intro
```

But introduction-only. No Elimination. It just exists.

Logical Constant True

True is simply true.

Get Proof for free:

```
def t : True := True.intro
```

But introduction-only. No Elimination. It just exists.

It's the unit type by CH. It is *uniquely* inhabited by the 0-tuple ().

Logical Constant False

Logical Constant False

Represents a contradiction. You shouldn't be able to obtain.

```
def f : False := -- impossible
```

Logical Constant False

Represents a contradiction. You shouldn't be able to obtain.

```
def f : False := -- impossible
```

Opposite of True: No introduction. Elimination-Only.

Logical Constant False

Represents a contradiction. You shouldn't be able to obtain.

```
def f : False := -- impossible
```

Opposite of True: No introduction. Elimination-Only.

Most powerful elimination. Anything follows from a contradiction.

```
false.elim :  $\forall (q : \text{Prop}), \text{False} \rightarrow q$ 
```

Logical Constant False

Represents a contradiction. You shouldn't be able to obtain.

```
def f : False := -- impossible
```

Opposite of True: No introduction. Elimination-Only.

Most powerful elimination. Anything follows from a contradiction.

```
false.elim : ∀ (q : Prop), False → q
```

If we include False as axiom in our type system, it becomes **unsound**.

```
axiom false : False
variable (q : Prop)
theorem hq : q := false.elim
```

Logical Constant False

Represents a contradiction. You shouldn't be able to obtain.

```
def f : False := -- impossible
```

Opposite of True: No introduction. Elimination-Only.

Most powerful elimination. Anything follows from a contradiction.

```
false.elim : ∀ (q : Prop), False → q
```

If we include False as axiom in our type system, it becomes **unsound**.

```
axiom false : False  
variable (q : Prop)  
theorem hq : q := false.elim
```

Is empty type by CH. The uninhabited type, that has no terms.

Logical Constant False

Represents a contradiction. You shouldn't be able to obtain.

```
def f : False := -- impossible
```

Opposite of True: No introduction. Elimination-Only.

Most powerful elimination. Anything follows from a contradiction.

```
false.elim : ∀ (q : Prop), False → q
```

If we include False as axiom in our type system, it becomes **unsound**.

```
axiom false : False
variable (q : Prop)
theorem hq : q := false.elim
```

Is empty type by CH. The uninhabited type, that has no terms.

Even though no introduction, still appears in formula.

```
example : p ∧ False ↔ False := Iff.intro
  (λ hpf → hpf.right)
  (λ hf → hf.elim)
```

Logical Negation $\neg$

Makes use of False

Logical Negation $\neg$

Makes use of False

Negation $\neg p$ in Lean is defined as $p \rightarrow \text{False}$.

Logical Negation $\neg$

Makes use of False

Negation $\neg p$ in Lean is defined as $p \rightarrow \text{False}$.

Elimination:

If you got a $hp : p$ and a $hnp : p \rightarrow \text{False}$ you would obtain a contradiction.

Logical Negation $\neg$

Makes use of False

Negation $\neg p$ in Lean is defined as $p \rightarrow \text{False}$.

Elimination:

If you got a $hp : p$ and a $hnp : p \rightarrow \text{False}$ you would obtain a contradiction.

We can proof how to get **contraposition** of implication.

```
example : (p → q) → (¬q → ¬p) :=  
  λ (hpq : p → q) ↪  
    λ hnq ↪ λ hp ↪ hnq (hpq hp)
```

Curry-Howard Isomorphism for Propositional Logic

Logic Name	Logic Notation	Type Notation	Type Name
Proposition	P	T	Type
Proof	p	t	Term
True	$\top$	$\top$	Unit Type
False	$\perp$	$\perp$	Empty Type
Implication	$\rightarrow$	$\rightarrow$	Function
Not	$\neg A$	$A \rightarrow \perp$	Function to Empty Type
And / Conjunction	$A \wedge B$	$A \times B$	Product Type
Or / Disjunction	$A \vee B$	$A + B$	Sum Type

Predicate Logic in Lean

Predicate Logic in Lean

Unary Predicate can be represented as

```
variable ( $\alpha$  : Type) (p :  $\alpha \rightarrow$  Prop)
```

Predicate Logic in Lean

Unary Predicate can be represented as

```
variable (α : Type) (p : α → Prop)
```

Given $x : \alpha$, then $p\ x$ denotes the assertion that p holds of x .

Predicate Logic in Lean

Unary Predicate can be represented as

`variable ( $\alpha$  : Type) (p :  $\alpha \rightarrow$  Prop)`

Given $x : \alpha$, then $p\ x$ denotes the assertion that p holds of x .

Binary Relation on α is $r : \alpha \rightarrow \alpha \rightarrow$ Prop.

Given $x\ y : \alpha$ then, $r\ x\ y$ denotes the assertion that x is related to y .

Predicate Logic in Lean

Unary Predicate can be represented as

`variable (α : Type) (p : α → Prop)`

Given $x : \alpha$, then $p\ x$ denotes the assertion that p holds of x .

Binary Relation on α is $r : \alpha \rightarrow \alpha \rightarrow \text{Prop}$.

Given $x\ y : \alpha$ then, $r\ x\ y$ denotes the assertion that x is related to y .

Type for **Universal Quantification**: $\forall x : \alpha, p\ x$

Type for **Existential Quantification**: $\exists x : \alpha, p\ x$

Predicate Logic in Lean

Unary Predicate can be represented as

```
variable (α : Type) (p : α → Prop)
```

Given $x : \alpha$, then $p\ x$ denotes the assertion that p holds of x .

Binary Relation on α is $r : \alpha \rightarrow \alpha \rightarrow \text{Prop}$.

Given $x\ y : \alpha$ then, $r\ x\ y$ denotes the assertion that x is related to y .

Type for **Universal Quantification**: $\forall x : \alpha, p\ x$

Type for **Existential Quantification**: $\exists x : \alpha, p\ x$

```
variable (men : Type) (barber : men)
```

```
variable (shaves : men → men → Prop)
```

```
example (h :  $\forall x : \text{men}, \text{shaves}\ \text{barber}\ x \leftrightarrow \neg \text{shaves}\ x\ x$ ) : False :=
```

```
  Classical.byCases
```

```
    (λ hsbbs ↦ absurd hsbbs ((h barber).mp hsbbs))
```

```
    (λ hnsbbs ↦ absurd ((h barber).mpr hnsbbs) hnsbbs)
```

Defining Mathematical Objects in Lean

```
structure Group (α : Type u) where
  op : α → α → α
  id : α
  inv : α → α
  op_assoc : ∀ a b c, op (op a b) c = op a (op b c)
  op_idl : ∀ a, op id a = a
  op_invl : ∀ a, op (inv a) a = id
```

Defining Mathematical Objects in Lean

```
structure Group (α : Type u) where
  op : α → α → α
  id : α
  inv : α → α
  op_assoc : ∀ a b c, op (op a b) c = op a (op b c)
  op_idl : ∀ a, op id a = a
  op_invl : ∀ a, op (inv a) a = id
```

Proofs in **Term-Mode** can get lengthy!

-- A group with a left inverse also has a right inverse.

```
theorem op_invr {α : Type u} (G : Group α) : ∀ a, G.op a (G.inv a) = G.id :=
  λ a ↦
    calc G.op a (G.inv a)
      _ = G.op G.id (G.op a (G.inv a)) := Eq.symm (G.op_idl (G.op a (G.inv a)))
      _ = G.op (G.op (G.inv (G.inv a)) (G.inv a)) (G.op a (G.inv a)) := congrArg (λ l ↦ G.op l (G.op a (G.inv a))) (Eq.symm (G.op_invl (G.inv a)))
      _ = G.op (G.inv (G.inv a)) (G.op (G.inv a) (G.op a (G.inv a))) := G.op_assoc (G.inv (G.inv a)) (G.inv a) (G.op a (G.inv a))
      _ = G.op (G.inv (G.inv a)) (G.op (G.op (G.inv a) a) (G.inv a)) := congrArg (G.op (G.inv (G.inv a))) (Eq.symm (G.op_assoc (G.inv a) a (G.inv a)))
      _ = G.op (G.inv (G.inv a)) (G.op G.id (G.inv a)) := congrArg (λ l ↦ G.op (G.inv (G.inv a)) (G.op l (G.inv a))) (G.op_invl a)
      _ = G.op (G.inv (G.inv a)) (G.inv a) := congrArg (G.op (G.inv (G.inv a))) (G.op_idl (G.inv a))
      _ = G.id := G.op_invl (G.inv a)
```

Tactic Mode

Different way of writing proofs using **Tactic-Mode**.

Here the **automated proof writing** of Lean comes into play.

This makes Lean an *Interactive* Theorem Prover.

```
theorem and_comm (p q : Prop) : p ∧ q → q ∧ p := by
  intro h          -- assume p ∧ q with proof h, the goal is q ∧ p
  apply And.intro   -- goal is split into two subgoals, one is q and the other is p
  · exact h.right    -- first subgoal is exactly the right part of h : p ∧ q
  · exact h.left     -- second subgoal is exactly the left part of h : p ∧ q
```

Inductive types

The natural numbers can be defined inductively.

Definition inspired by **Peano Axioms**.

```
inductive Nat : Type
```

```
| zero : Nat
```

```
| succ : Nat → Nat
```

Inductive types

The natural numbers can be defined inductively.

Definition inspired by **Peano Axioms**.

```
inductive Nat : Type
| zero : Nat
| succ : Nat → Nat
```

Addition of natural numbers can be defined recursively, using **pattern matching**.

```
def Nat.add : Nat → Nat → Nat
| n, Nat.zero    => n                --  $n + 0 = n$ 
| n, Nat.succ m => Nat.succ (Nat.add n m) --  $n + \text{succ}(m) = \text{succ}(n + m)$ 
```

Constructive vs. Non-Constructive Math

There are two kinds of logical arguments.

Constructive vs. Non-Constructive Math

There are two kinds of logical arguments.

Is there an $x > 2$ such that $2^x = x^2$?

Constructive vs. Non-Constructive Math

There are two kinds of logical arguments.

Is there an $x > 2$ such that $2^x = x^2$?

- Constructive: Yes, 4.
- Nonconstructive: Yes, because Intermediate Value theorem.

Constructive vs. Non-Constructive Math

There are two kinds of logical arguments.

Is there an $x > 2$ such that $2^x = x^2$?

- Constructive: Yes, 4.
- Nonconstructive: Yes, because Intermediate Value theorem.

Constructive Proofs are better!

Example: $P = NP$.

Constructive vs. Non-Constructive Math

There are two kinds of logical arguments.

Is there an $x > 2$ such that $2^x = x^2$?

- Constructive: Yes, 4.
- Nonconstructive: Yes, because Intermediate Value theorem.

Constructive Proofs are better!

Example: $P = NP$.

But classical logic can proof more statements.

Lean by computational nature is constructive, but classical logic also possible.

Constructive vs. Non-Constructive Math

There are two kinds of logical arguments.

Is there an $x > 2$ such that $2^x = x^2$?

- Constructive: Yes, 4.
- Nonconstructive: Yes, because Intermediate Value theorem.

Constructive Proofs are better!

Example: $N = NP$.

But classical logic can proof more statements.

Lean by computational nature is constructive, but classical logic also possible.

Governed by a single axiom: The **Law of Excluded Middle** (EM)

`Classical.em (p : Prop) : p ∨ ¬p`

Constructive vs. Non-Constructive Math

There are two kinds of logical arguments.

Is there an $x > 2$ such that $2^x = x^2$?

- Constructive: Yes, 4.
- Nonconstructive: Yes, because Intermediate Value theorem.

Constructive Proofs are better!

Example: $\mathbb{N} = \text{NP}$.

But classical logic can proof more statements.

Lean by computational nature is constructive, but classical logic also possible.

Governed by a single axiom: The **Law of Excluded Middle** (EM)

`Classical.em (p : Prop) : p ∨ ¬p`

This now opens the door for many thing we take for granted.

- Proof by Contradiction and Proof by Case Distinction
- Double Negation $\neg\neg p \equiv p$

Only this makes logic binary! Denk mal drüber nach...

Mathlib Showcase

<https://leanprover-community.github.io/mathlib-overview.html>

https://leanprover-community.github.io/mathlib4_docs/

Write Lean Code yourself

- Online on
<https://live.lean-lang.org/>
- In VsCode using the extension “Lean 4”

Further Reading

- Lean Book “Theorem Proving in Lean4”
https://leanprover.github.io/theorem_proving_in_lean4
- Another Lean Book “Logic and Proof”
https://leanprover-community.github.io/logic_and_proof
- YT: “Why Vlad Tenev and Tudor Achim of Harmonic Think AI Is About to Change Math—and Why It Matters”
<https://youtu.be/NvAxuCIBb-c?si=Nhs6o-79xwwMKA2Z>
- YT: “Type Theory for the Working Rustacean - Dan Pittman” <https://youtu.be/BdXWlQsd7RI?si=7ZhTNCOI6e3P12ds>
- Beyond “just” Type Theory: Homotopy Type Theory (HoTT) 🧠
<https://youtube.com/playlist?list=PL245PKGUDdcN9-El9D7DRefwX4c9feiYq&si=RnAo2CUVPXfPNqkf>

Thank U :)