Theorem Proving in Lean4

Reference:

Theorem Proving in Lean 4 - Theorem Proving in Lean 4 (leanprover.github.io)

Lean as a programming language

Define a term using following syntax:

def [identifier] [context] : [type] := ……

Use the following format to write a function:

fun x1 … xn => ……

“#check” command returns the type of expressions :

#check [expressions]

“#eval” command evaluates the expressions we defined

#eval [expressions]

“Nat -> Nat -> Nat” type is a function that takes in two variables.
It equals to “Nat -> (Nat -> Nat)”. In the example it means after taking in the first parameter “1”, “g 1” is a function of type “Nat -> Nat”
“g” and “g’” behave the same.

```
  def a: Float :=6.7
  def b: Int := -6
  def s: String:="abc"
  def l: List Nat :=[1,2,3,4]
  def f: Nat->Nat := fun x => x+1
  def g: Nat->Nat->Nat := fun x y => x+y
  def g' (n:Nat) : Nat -> Nat := fun x=> x+n
  def h: Nat->Nat:= fun x => if x<10 then x else 10

I #check a      ■ a : Float
I #check b     ■ b : Int
I #check s     ■ s : String

I #eval f 5     ■ 6

I #eval g 4 5     ■ 9
I #check g'      ■ g' (n : Nat) : Nat → Nat
I #eval g 4 5     ■ 9
I #check g' 4     ■ g' 4 : Nat → Nat
I #check g 1     ■ g 1 : Nat → Nat

I #eval h 11     ■ 10
I #eval h 1     ■ 1
```

Theorem proving in Lean

Type system in Lean is quite flexible and expressive.

In fact, each proposition is a type, and to prove a proposition is just to construct a term of this type.

\forall \exists \< \>

"theorem" is another name for "def"

Proposition: For all natural number a, there exists a b so that b>a+1.

```
  theorem th1 : ∀a:Nat, ∃ b, b> a+1 :=
E   fun a => ⟨a+2, by ⟩      ■■ unsolved goals  a : Nat  ⊢ a + 2 > a + 1
```

This proposition is proved if we can construct a function that takes in any natural number a and spits out a prove that "there exists a b so that b>a+1". To prove "there exists a b so that b>a+1", we actually need to construct a pair <b, proof>, where b is the number we have found and "proof" is the proof that "b > a+1".

```
  theorem th1 : ∀a:Nat, ∃ b, b > a+1 :=
    fun a => ⟨a+2, by simp ⟩

I #check th1      ■ th1 (a : Nat) : ∃ b, b > a + 1

  theorem th1': (a:Nat) -> (∃b, b > a+1) :=
    fun a => ⟨a+2, by simp ⟩

I #check th1'     ■ th1' (a : Nat) : ∃ b, b > a + 1
```

The goal is reduced to prove "a+2>a+1". I used the tactics "simp" for simplifying. "by" keyword starts a section of tactics.

Dependent types

```
I #check (A:Type)->(A->A)      ■ (A : Type) → A → A : Type 1
  def term1:(A:Type)->(A->A) := fun _ a=>a
  def term1':{A:Type}->(A->A):= fun a =>a
I #check term1      ■ term1 (A : Type) : A → A
I #check term1'     ■ term1' {A : Type} : A → A

I #check  (A:Type)×(A->A)      ■ (A : Type) × (A → A) : Type 1
  def term2:(A:Type)×(A->A) := ⟨Nat,fun x=>x+1 ⟩
I #check term2      ■ term2 : (A : Type) × (A → A)
```

In Lean dependent types are functions or pairs where the type of later variables can depend on the previous variables.

(a:Nat)->(a>0) is a proposition that for all natural number a, a>0.
(b:Nat)×(b>0) is a proposition that there exists a natural number b, such that b>0.
(But this syntax is invalid. We write ∃b,b>0. When the type of last variable is a proposition, we use “∃” expressions instead.)

We can leave “_” in expressions to ask Lean to fill in this whole automatically.
“{A:Type}” in the example above asks Lean to infer “A” automatically.
<…> is the anonymous constructor.

Variable declaration and namespaces

Use the keyword “variable” to declare variables without defining them:

variable (a1:[Type]) … (an:[Type])

This declaration is valid inside a namespace:

namespace [name]

…

end [name]

Inside a namespace, declared variables can be used directly in definitions and proofs.
To call definitions inside a namespace, we should write:

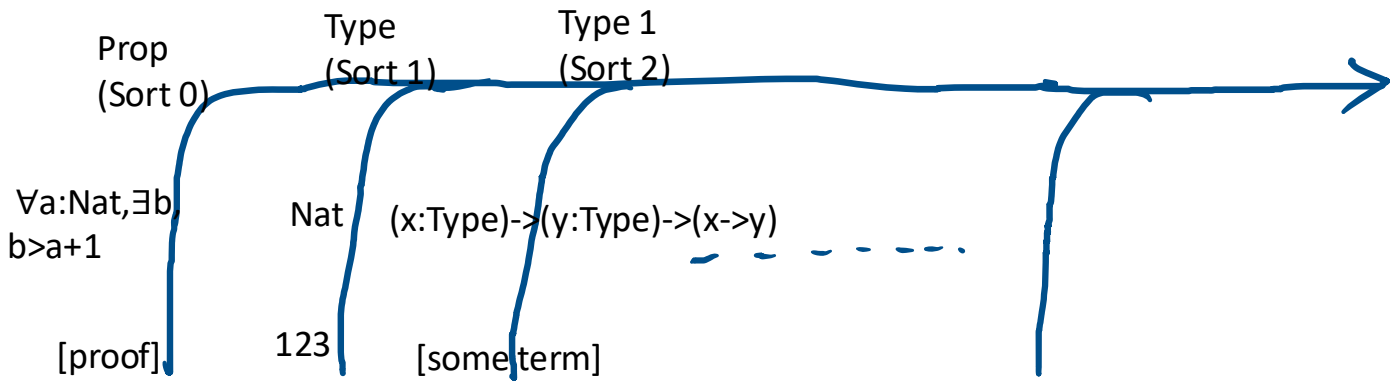
[namespace].[identifier]

If we call a definition inside a namespace, the variables declared inside this namespace that are used by this definition will be automatically added to this definition as context.

```
  variable (v1:Nat) (v2:Nat->Nat)
I #check v1      ■ v1 : Nat
I #check v2      ■ v2 : Nat → Nat

  namespace ns1

  variable (v3:Int) (T:Type)
I #check v3      ■ v3 : Int
I #check v1      ■ v1 : Nat
  def f2 :T->T := fun x=> x
I #check f2      ■ ns1.f2 (T : Type) : T → T

  end ns1

I #check ns1.f2     ■ ns1.f2 (T : Type) : T → T
I #eval ns1.f2 Nat 4     ■ 4
I #check v1      ■ v1 : Nat
E #check f2      ■ unknown identifier 'f2'
E #check v3      ■ unknown identifier 'v3'
```

Type system in Lean

```
I #check (∀a:Nat,∃b,b>a+1)     ■ ∀ (a : Nat), ∃ b, b > a + 1 : Prop
I #check Prop     ■ Prop : Type
I #check Type     ■ Type : Type 1
I #check Type 1     ■ Type 1 : Type 2
I #check Type _     ■ Type u_1 : Type (u_1 + 1)
I #check Sort 0     ■ Prop : Type
I #check Sort 1     ■ Type : Type 1
I #check Sort 2     ■ Type 1 : Type 2
I #check (x:Type 6)->(y:Type 3)->(x->y)->Nat     ■ (x : Type 6) → (y : Type 3) → (x → y) → Nat : Type 7
I #check (a:Type)->(b:Type)->(1>3)     ■ Type → Type → 1 > 3 : Prop

  variable (T:Type 4) (termT:T)
E variable (a:termT)     ■ type expected, got    (termT : T)
```



The type system has the hierarchy shown on the left. “Type n” is said to have universe level n.
A dependent type (t1:Type u1)-> … ->(tn:Type un) typically has type “Type max{u1+1, … ,un+1}”.
However, if “tn” is of type “Prop”, the whole dependent type will be of type “Prop”.

Inductive types

```
  inductive myList (A:Type)  : Type where
  |c1:(a:A)->myList A
  |c2:(myList A)->(a:A)->myList A

I #check ((myList.c1 4).c2 6).c2 5     ■ ((myList.c1 4).c2 6).c2 5 : myList Nat

I #eval Nat.zero     ■ 0
I #eval Nat.zero.succ.succ.succ.succ     ■ 4
```

((myList.c1 4).c2 6).c2 5 is the list [4, 6, 5]

All user defined types in Lean are inductive types.

We can construct inductive types using following syntax:

inductive [name] (a1:[type]) … (an:[type]) :Sort u where

|constructor_1: ….->[name] a1 … an

….

|constructor_m: ….->[name] a1 … an

a1, … , an should be types instead of terms, and they should be able to be inferred from the input of constructors.

The “Sort u” specification of the universe level of this inductive type should be strictly greater that the universe level of constructors.

A constructor can construct a term of the inductive type it belongs to.

We can use “[name].constructor” to call a constructor. We can also use “[term].constructor” when the “[term]” is of this inductive type and the constructor we are calling takes in a term of this inductive type.

inductions

Because all user defined types in Lean are inductive types and all terms are constructed by constructors, we can do inductions and case distinctions on (a:A) simply by looking at what constructor is used to construct the term a.

Define a case distinction using following syntax:

```
match [term] with
 |constructor_1 … => …
 …
 |constructor_n … => …
```

In the example, “moreThanOne”, “last” and “first” is defined under the namespace “myList”, so we can use [term].moreThanOne etc. to call these definitions.
Recursion is used in the definition of “myList.first”.
When using recursion, Lean will check whether this recursion can be proved to terminate.
Recursion is important in many proofs involving natural numbers.

```
def myList.moreThanOne {A:Type}:myList A -> Nat :=
  fun l =>
  match l with
    |.c1 _ => 0
    |.c2 _ _ => 1

def myList.last {A:Type}:myList A -> A :=
  fun l =>
  match l with
    |.c1 a => a
    |.c2 _ a => a

def myList.first {A:Type}:myList A -> A :=
  fun l =>
  match l with
    |.c1 a => a
    |.c2 l' _ => l'.first

I #eval (((myList.c1 4).c2 6).c2 5).moreThanOne     ■ 1
I #eval (((myList.c1 4).c2 6).c2 5).last     ■ 5
I #eval (((myList.c1 4).c2 6).c2 5).first     ■ 4
```

inductions

Example: (Not the best proof)

```
theorem th4 (a b:Nat) (h1:2<=a) (h2:2<=b): a+b <=a*b :=
  match a with
  |.zero => by simp at h1
  |.succ .zero => by simp at h1
  |.succ (.succ .zero)=> by
    simp
    calc
    2+b <= b+b := by simp [h2]
    _ = 1*b+1*b := by simp
    _ = _ := by rw [← Nat.right_distrib ]
  |.succ (.succ (.succ a''')) => by
    let a':=a'''.succ.succ ;have p1: a'=a'''.succ.succ := by rfl
    have h3: 2<= a':= by simp [p1]
    rw [p1.symm]
    replace h3:=th4 _ _ h3 h2
    simp;rw [Nat.right_distrib];simp
    calc
    _ <= a'+2:= by simp
    _ <= a'+b:= by simp [h2]
    _ <= _ := h3
```

We can use the following syntax anywhere in our proof or definition of define a local variables for substitutions

let [identifier]:[type]:= …

It can also be used outside tactics mode.

Structures

Structures are a variation of inductive type:

structure [name] [parameter] where
[field1]: …
[field2]: …

We can define a term of this structure using the anonymous constructor:

[identifier]:[Type]:=
<field_1, … , field_n>

Or

{field_1:= …, …, field_n:= …
:[Type]}

```
structure myGroup (S:Type) where
  mul: S->S->S
  e:S
  id: ∀x:S, (mul e x = x) ∧ (mul x e =x)
  inv: ∀x:S, ∃x':S, (mul x x' = e) ∧ (mul x' x = e)
  assoc: ∀ x y z :S, mul (mul x y) z = mul x (mul y z)

def myG:myGroup Int := ⟨
  fun x y => x+y,0,
  by intro x; simp,
  by intro x ;refine' ⟨-x,_ ⟩; simp [Int.add_right_neg,Int.add_left_neg]  ,
  Int.add_assoc⟩

I #print myG     ■ def myG : myGroup Int :=  { mul := fun x y => x + y, e := 0, i

variable (T:Type) (myG2:myGroup T)

I #check myG2.assoc     ■ myG2.assoc : ∀ (x y z : T), myG2.mul (myG2.mul x y) z =
```

Classes and instances

Classes are a variation of structures.
However, these structures are anonymous. We need to declare an instance to use methods (fields of structures) under these classes.
Methods can be overloaded when used on different terms.
We sometimes see functions of the form “{T}->[inst]->(xxxx)-> … ->(xxxx)”. “{T}” asks Lean to infer a type “T”, and “[inst]” asks Lean to infer an instance.

Axioms and sorry

```
namespace wrong

axiom w:1+1=3

theorem thw: 5+7=0 := by
  have h1:=w
  simp at h1

end wrong

W theorem ths : 5+7=0:= by sorry     ■ declaration uses 'sorry'
```

Declare an axiom using following syntax:

axiom [identifier]:[Type]

The “sorry” tactic can fill in the proof of any proposition

Using tactics

"by" keyword starts a section of tactics.
Tactics are programs that help to construct proof terms automatically.

Some useful tactics :

have h:[type]:=	a tactic version of "let"
have h:[type]	declare a local goal without proving it
rw [xxx,<-xxx,…]	rewrite the goal
rw [xxx,…] at h	rewrite expressions or local goal h
simp	simplify
simpa	simplify all
simp at h	
simpa using h	
simp_rw	used to rewrite expressions when motive is different
intro	introduce a variable in the statement
\t	substitution
congr	reduce f a = f b to a=b
exact p	finish a proof with p
by_cases	discuss the two cases where a hypothesis is true or not
Refine' xxx _ xxx _ ….	refine the goal with xxx but leave some "wholes" to fill in
induction'	use induction to argue
suffice	
calc	do calculations in steps

Using tactics

In Mathlib:

ring	use ring property to simplify
abel	use abelian group property to simpify
zify	convert a natural number proposition into integer proposition
push_neg	push negation into a proposition
contrapose	argue contrapositively
cases’	used to discuss AorB type proposition
by_contra …	argue by contradiction

Recursion can also be used in tactics mode, which means we can call a theorem or definition itself when constructing local variables or statements, as long as Lean can figure out the proof that this recursion can terminate.