Theorem Prover on Analysis Topics

Yuxuan Sun

Spring 2022, Analysis 2

Contents

1 Introduction

Mathematical proofs could get more difficult to read and verify as they grow to be more complicated. What else could we rely on to verify the proofs are correct besides trust on each other's professionalism? Building on homotopy type theory, automated theorem proving has the ability to verify our proofs. As there are many modern theorem provers, such as Agda, Lean, Coq, the intention of the paper is to give a brief introduction on how to construct basic analysis definitions and to prove theorems using *Lean*, with sufficient explanations for mathematicians on how the language works.

When preparing for this paper, I mainly rely on [1] and [3]. The choice of using *Lean* is based on [2]. Since the paper in general comes from understanding the mixture of the sources and the online library of *Lean*, I won't be able to cite them separately in below.

We will start with some very mathematically simple definitions and proofs, then move to a sequence-limit proof in Analysis I. Due to page limit, I will briefly introduce what the proofs related to functional analysis look like at the end.

2 Basic Definition and Proofs

Specifically, let's look at the definition of upper bounds written in Lean.

```
def upperBounds (A : set ℝ) := { x : ℝ  | ∀ a ∈ A,  a ≤ x }
```

Here we defined a function named `upperBounds`, which takes one input: `A` whose **type** is a set of real numbers. The output of the function is a set (indicated by { }, and the type of elements `x` in the set is real numbers. `x` also need to satisfy the property that for all `a` ∈ `A`, `a` ≤ `x`.

We could also define the maximum element in a set as the following.

```
def isMaximum (a : ℝ) (A : set ℝ) := a ∈ A ∧ a ∈ upperBounds A
```

This function **isMaximum** takes two inputs: `a` whose type is real number; `A` whose type is a set of real numbers.

Its output is a boolean: `a` is in `A` **and** `a` is in upperBounds `A`. Notice that here we called the function we defined above so upperBounds A is the output of the function defined above.

An observation is that since the logical connection here is **and**, if **isMaximum a A** returns true, we are guaranteed two truth statements: a is in A and a is in upperBounds A.

Now we could prove something!

Let's prove that if both x and y are maximum of A, then $x = y$

2.1 details of the first proof

First a longer and more detailed proof is the following.

```
lemma uniqueMax (A : set ℝ) (x y : ℝ)
(hx : isMaximum x A) (hy : isMaximum y A)
: x = y
:=
begin
        cases hx with x_in x_up,
        cases hy with y_in y_up,
        specialize x_up y,
        specialize x_up y_in,
        specialize y_up x x_in,
        linarith,
end
```

The lemma takes four assumptions:

1. `A` whose type is a set of real numbers

2. `x,y` whose types are real numbers

3. `hx` whose type is a boolean derived from `isMaximum` above. Recall that then it contains two statements:

- **a.** x is in A
- **b.** x is in upperBounds A (x is bigger than anything in A)

4. `hy` is similar as `hx`

Assumptions are usually stated before colon **:**

After the colon it's the result we want to prove, namely: $x = y$

:= indicates that our proof begins.

Now let's look at the details of the proof.

At line 6, we have `cases ... with ...`, operations like this are called tactics in *Lean*. It splits the hypothesis `hx` into two pieces, named `x_in` and `x_up`, corresponding to **3a** and **3b** respectively, because `hx` is a conjunction, it will be easier for us to use.

Similarly for line 7.

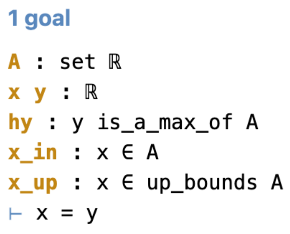
At line 8, `specialize` is a tactic which applies everything following the first thing to the first thing. In this line, it gives the hypothesis `x_up` the number `y`, so now our `x_up` is renewed as: `y ∈ A → y ≤ x`

At line 9, in order to have `y ≤ x`, we need to give our new `x_up` the precedent: `y` is in `A`, which is provided by `y_in`. After line 9, our proof so far has shown that $y \le x$.

At line 10, it's just a quicker way to write line 8 and 9 together.

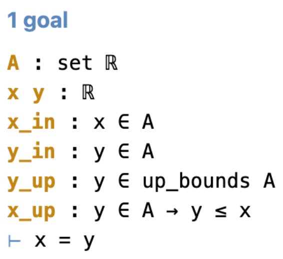
`linarith` is a basic linear arithmetic package lean has that allow us to turn $x \le y$ and $y \le x$ into $x = y$. Thus our proof is done.

One feature *Lean* has is that as you are writing proofs, it could tell you how your hypothesis has been updated, as shown below.



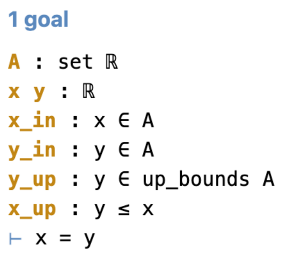
```
1 goal
A : set ℝ
x y : ℝ
hy : y is_a_max_of A
x_in : x ∈ A
x_up : x ∈ up_bounds A
⊢ x = y
```

Figure 1: At line 6



```
1 goal
A : set ℝ
x y : ℝ
x_in : x ∈ A
y_in : y ∈ A
y_up : y ∈ up_bounds A
x_up : y ∈ A → y ≤ x
⊢ x = y
```

Figure 2: At line 8



```
1 goal
A : set ℝ
x y : ℝ
x_in : x ∈ A
y_in : y ∈ A
y_up : y ∈ up_bounds A
x_up : y ≤ x
⊢ x = y
```

Figure 3: At line 9

2.2 simplified first proof

Certainly, for a simple proof, we could also write a shorter version (next page).

`have from` is a tactic which produce a hypothesis by applying what's following from. Here `hy.2` refers to the second part of `hy`, which is `y ∈ upperBounds A`.

The advantages of writing proofs in this way is straightforward: it's shorter. However, it does decrease the readability of the proof, as it's hard for readers to know what `hy.2` refers to.

```
lemma uniqueMax (A : set ℝ) (x y : ℝ)
(hx : isMaximum x A) (hy : isMaximum y A)
: x = y
:=
begin
        have : x ≤ y, from hy.2 x hx.1,
        have : y ≤ x, from hx.2 y hy.1,
        linarith,
end
```

3 Sequence-Limit Definition and Proof

Let's prove another thing in analysis, namely, given two sequences u_n, v_n, if $u_n \to l$ and $v_n \to l'$ then $(u_n + v_n) \to (l + l')$

Before that, we need the definition of sequence-limit.

```
def seq_limit (u : ℕ → ℝ) (l : ℝ) : Prop :=
∀ϵ > 0,∃ N, ∀ n ≥ N, |u n - l| ≤ ϵ
```

Let's go straight to the proof and explain the details later.

```
1   lemma sequenceAdd (hu : seq_limit u l1) (hv : seq_limit v l2) :
2   seq_limit ( u + v) (l1 + l2)
3   :=
4   begin
5           intros ϵ ϵ_pos,
6           cases hu (ϵ/2) (by linarith) with N1 hN1,
7           cases hv (ϵ/2) (by linarith) with N2 hN2,
8           use max N1, N2,
9           intros n hn,
10          rewrite ge_max_iff at hn,
11          calc
12          | (u + v) n - (l1 + l2) | = | un + vn - (l1 + l2) | :rfl
13          . . . = | (u n - l1) + (v n - l2) | : by congr 1; ring
14          . . . ≤ | un - l1 | + | vn - l2 | : by apply abs_add
15          . . . ≤ ϵ : by linarith [hN1 n (by linarith), hN2 n (by linarith)]
16  end
```

The places with `intros` and `cases` are similar as our basic example. It a more complicated usage but the ideas are similar: `intros` give names to what we had and `cases` splits the hypothesis so that we could use them later.

In line 7, we use a library function named `max` to take the maximum of `N1, N2`. Later in line 8, we use `n` to denote the number and `hn` to denote the statement that `n` ≥ `max N1 N2`.

In line 8, we rewrite `hn` using a library function named `ge_max_iff`, which just split `hn` into `n` ≥ `N1` ∧ `n` ≥ `N2`.

In line 8, since it's just splitting and renaming, we could also write

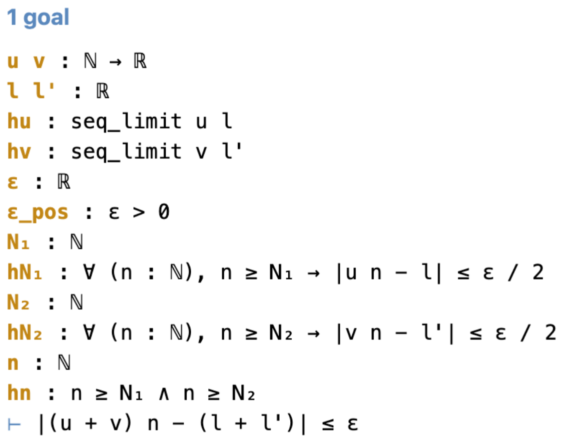
```
cases ge_max_iff.mp hn with hn1 hn2
```

Starting at line 11, one could just consider it as an align environment in latex with reasons mandatorily required. Whatever is following `by` are library theorems, whose meanings one could probably guess.

Notice that at line 15, the square bracket after linarith is a quicker way to write the following.

```
have fact1 :  |u n - l1| ≤ ϵ/2,
    from hN1 n (by linarith),
have fact2 : |v n - l2| ≤ ϵ/2,
    from hN2 n (by linarith),
```

For reference, before line 11, what we built up is shown in Figure 4.



```
1 goal
u v : ℕ → ℝ
l l' : ℝ
hu : seq_limit u l
hv : seq_limit v l'
ε : ℝ
ε_pos : ε > 0
N₁ : ℕ
hN₁ : ∀ (n : ℕ), n ≥ N₁ → |u n - l| ≤ ε / 2
N₂ : ℕ
hN₂ : ∀ (n : ℕ), n ≥ N₂ → |v n - l'| ≤ ε / 2
n : ℕ
hn : n ≥ N₁ ∧ n ≥ N₂
⊢ |(u + v) n - (l + l')| ≤ ε
```

Figure 4: Before line 11 (cursor at line 10)

4 Functional Analysis

Recall what we learnt in class about functions in L^1.

Theorem 4.1 **($L^1(\mathbb{R})$ closed under addition).** Given $f_1, f_2 \in L^1(\mathbb{R})$, then $f_1 + f_2 \in L^1(\mathbb{R})$, namely

$$\int_{-\infty}^{\infty} f_1(x) + f_2(x) dx = \int_{-\infty}^{\infty} f_1(x) dx + \int_{-\infty}^{\infty} f_2(x) dx$$

The statement of it is the following. `nnnorm` here is the notation we use when

```
lemma lintegral_nnnorm_add
  {f : a → b} {g : a → y}
  (hf : ae_strongly_measurable f u)
  (hg : ae_strongly_measurable g u)
  : ∫⁻ a, nnnorm (f a) + nnnorm (g a) du =
            ∫⁻ a, nnnorm (f a) du + ∫⁻ a, nnnorm (g a) du
```

talk about functions.

Due to the page limit and the length of the proof (it could be really short by using a bunch of library functions but really long if we spell everything out). The proof is omitted. It is long mainly because that we need to seriously take about measures.

The takeaway here may be my failed attempt of writing it in the "hand-wavy" way we did in class, where we skipped the detailed discussions on measure. I have not figured out a way to do so such that the prover would accept.

5 Discussion

The power of using theorem prover is obvious: it's an insurance of our mathematical correctness. However as I go along the process my major thought is: is it worth it?

Using theorem prover, especially one with sufficient libraries to use, shares the deficit of all other programming languages, it could be hard to read sometimes if the code is abbreviated. However, advanced mathematical proofs also have this problem, as sometimes it could even be hard to understand when the content is the same but notations are different. In theorem prover, at least one could keep tracing back until one finds the source code that the writer is using.

In conclusion, I hope this paper gives a comprehensive introduction on theorem prover to mathematicians who are not familiar with programming languages at all, and maybe some motivations to try to use it in some of their works.

References

[1] Jeremy Avigad, Leonardo de Moura **and** Soonho Kong. “Theorem Proving in Lean”. **in**(**january** 2016): DOI: 10.1184/R1/6492902.v1.

[2] Sylvie Boldo, Catherine Lelay **and** Guillaume Melquiond. “Formalization of real analysis: a survey of proof assistants and libraries”. en. **in***Mathematical Structures in Computer Science*: 26.7 (**october** 2016), **pages** 1196–1233. DOI: 10.1017/S0960129514000437.

[3] The Univalent Foundations Program. “Homotopy Type Theory: Univalent Foundations of Mathematics”. en. **in***arXiv:1308.0729 [cs, math]*: (**august** 2013). arXiv: 1308.0729.