A formalisation of transcendence of e

Jujian Zhang

July 19, 2020

Contents

Abstract

The objective of this report is to present formalisations of some basic theorems from transcendental number theory with `Lean` and `mathlib` in the hope that it will serve as a motivation for mathematicians to be more curious about interactive theorem proving. The following theorems are formalised:

1. the set of algebraic numbers is countable, hence transcendental number exists:

```
theorem algebraic_set_countable : set.countable algebraic_set
theorem transcendental_number_exists : ∃ x : ℝ, transcendental x
```

2. all Liouville's numbers are transcendental:

```
theorem liouville_numbers_transcendental :
  ∀ x : ℝ, liouville_number x → transcendental x
```

3. $\alpha := \sum_{i=0}^{\infty} \frac{1}{10^{i!}}$ is a Liouville's number hence α is transcendental.

```
theorem liouville_α : liouville_number α
theorem transcendental_α : transcendental α :=
  liouville_numbers_transcendental α liouville_α
```

4. e is transcendental:

```
theorem e_transcendental : transcendental e
```


Disclaimer

The plan is to a self-contained report so that after chapter 2 any reader even without prior exposure to interactive theorem proving will be able to understand 3 where the details of formalisations and proofs reside. This should be relatively straightforward since the author is a `Lean`-dilettantes at best with only a partial picture of the full language. For the same reason, much of the code is perhaps not idiomatic or even plainly bad, thus it is not advisable to use this as a tutorial.

Chapter 1

Overview

1.1 Interactive theorem proving

Around 1920s, the German mathematician David Hilbert put forward the Hilbert programme to seek:

1. an axiomatic foundation of mathematics;
2. a proof of consistency of the said foundation;
3. Entscheidungsproblem: an algorithm to determine if any proposition is universally valid given a set of axioms.

The first two aims were later proved to be impossible by Gödel and the celebrated incompleteness theorems. Via the completeness of first order logic, the Entscheidungsproblem can also be interpreted as an algorithm for producing proofs using deduction rules. Even without a panacea approach for mathematics, computer still bears advantages against a carbon-based mathematician. Perhaps the most manifested advantage is the accuracy of a computer to execute its command and to recall its memories. Thus came the idea of **interactive theorem proving** — instead of hoping a computer algorithm to spit out some unfathomable proofs, assuming computers are given the ability to check correctness of proofs, human-comprehensible proofs can be verified by machines and thus guaranteed to be free of errors. With a collective effort, all theorems verified this way can be collected in an error-free library such that all mathematicians can utilise to prove further theorems which can then be added to the collection, ad infinitum [Boy+94]. Curry-Howard isomorphism provided the crucial relationship between mathematical proofs and computer programmes, more specifically relationship between propositions and types, to make such project feasible [KK11]. The idea will be explained in section 2 along with `Lean`.

The proof of "Kepler's conjecture[1]" will serve as an illustrative example of utility of interactive theorem proving. As early as 1998, Thomas Hales had

[1] the most efficient way to pack spheres should be hexagonally

claimed a proof [Hal98; HUW14], however the proof is controversial in the sense that mathematician even with great effort could not guarantee its correctness. A collaborative project using `Isabelle`[2] and `HOL Light`[3] verified the proof around 2014 and hence settled the controversy in 2017 [Hal+17]. There is also Georges Gonthier with his teams using `Coq`[4] who formalised the four colour theorem and Feit-Thompson theorem where the latter is a step to the classification of simple groups [Gon08; Gon+13]. Using `Lean`[5], Buzzard, Commelin, and Massot were able to formalise modern notion of perfectoid spaces [BCM20].

1.2 History of transcendental numbers

"Transcendence" as a mathematical jargon first appeared in a Leibniz's 1682 paper where he proved that sin is a transcendental function in the sense that for any natural number n there does not exist polynomials $p_0, \cdots, p_n$ such that

$$p_0(x) + p_1(x)\sin(x) + p_2(x)\sin(x)^2 + \cdots + p_n(x)\sin(x)^n = 0$$

holds for all $x \in \mathbb{R}$ [Bou98]. The Swiss mathematician Johann Heinrich Lambert in his 1768 paper proved the irrationality of e and π where he also conjectured their transcendence [Lam04]. It is until 1844 that Joseph Liouville proved the existence of any transcendental numbers and until 1851 an explicit example of transcendental number is actually given by its decimal expansion:[Kem16]

$$\sum_{i=1}^{\infty} \frac{1}{10^{i!}} = 0.11000100000\cdots.$$

However, this construction is still artificial in nature. The first example of a real number proven to be transcendental that is not constructed for the purpose of being transcendental was e. Charles Hermite proved the transcendence of e in 1873 with a method applicable with help of symmetric polynomial to transcendence of π in 1882 and later to be generalised to Lindemann-Weierstrass theorem in 1885 stating that if $\alpha_1, \cdots, \alpha_n$ are distinct algebraic numbers then $e^{\alpha_1}, \cdots, e^{\alpha_n}$ are linearly independent over the algebraic numbers [Bak90]. The transcendence of π was particularly celebrated because it immediately implied the impossibility of the ancient greek question of squaring the circle, i.e. it is not possible to construct a square, using compass and ruler only, with equal area to a circle. For this question is plainly equivalent to construct $\sqrt{\pi}$ which is not possible for otherwise π is algebraic. Georg Cantor in 1874 proved that algebraic numbers are countable hence not only did transcendental numbers exist, they exist in a ubiquitous manner – there is a bijection from the set of all transcendental numbers to $\mathbb{R}$ [Can32; Can78].

[2] a theorem prover relies extensively on dependent type theory and Curry-Howard correspondence.
[3] ibid.
[4] ibid.
[5] ibid.

In 1900, Hilbert proposed twenty-three questions, the 7th of which is regarding transcendental numbers: Is a^b transcendental, for any algebraic number a that is not 0 or 1 and any irrational algebraic number b? The answer is yes by Gelfond-Schneider theorem in 1934 [Gel34]. This has some immediate consequences such that

1. $2^{\sqrt{2}}$ and its square root $\sqrt{2}^{\sqrt{2}}$ are transcendental;
2. e^π is transcendental for $e^\pi = \left(e^{i\pi}\right)^{-i} = (-1)^{-i}$;
3. $i^i = e^{-\frac{\pi}{2}}$ is transcendental etc.

In contrast, none of $\pi \pm e$, $\pi e, \frac{\pi}{e}$, π^π, π^e etc are proven to be transcendental. It is also conjectured by Stephen Schanuel that given any n $\mathbb{Q}-$linearly independent $z_1, \cdots, z_n \in \mathbb{C}$, then $\operatorname{trdeg}\left(\mathbb{Q}(z_1, \cdots, zn, e^{z_1}, \cdots, e^{z_n})/\mathbb{Q}\right)$ is at least n [Lan66]. If this were proven, the algebraic independence of e and π would follow immediately by setting $z_1 = 1$ and $z_2 = \pi i$ with Euler's identity.

Chapter 2

Brief introduction to `Lean`

`Lean` is developed by Leonardo de Moura at Microsoft Research Redmond from 2013 using dependent type theory and calculus of inductive constraint [AMK15]. In this chapter, basic ideas of Curry-Howard isomorphism will be demonstrated by some basic examples of mathematical theorem expressed in `Lean` using dependent type theory.

2.1 Simple type theory

Unlike set theory where everything from natural numbers to modular forms is essentially a set. Type theory associate every expression with a `type`. In set theory, an element can belongs to different sets, for example 0 is simultaneously in $\mathbb{N} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$. However an expression can only have one type. 0 without any context will have type $\mathbb{N}$ and, to specify the zero with type $\mathbb{R}$ we write $(0 : \mathbb{R})$. If a has type α, we write $a : \alpha$. By a universe of types we mean a collection of types.Types can be combined to form new types in the following way:

- let α and β be types then $\alpha \to \beta$ is the type of functions from α to β: the element of type $\alpha \to \beta$ is a function that for any element of α gives an element of β. For mathematician this loosely means that for any two classes α and β, there is a new class $\hom(\alpha, \beta)$. Sometimes we are not bothered to give a function a name, we can use the λ notation: $(\lambda x : \alpha, \text{expression})$ has type $\alpha \to \ldots$ depending on the content of expression. This can be thought of $\mapsto$. For example $(\lambda x : \mathbb{N}, x + 1) : \mathbb{N} \to \mathbb{N}$.
- let α and β be types then $\alpha \times \beta$ is the cartesian product of α and β: the element of type $\alpha \times \beta$ is an ordered tuple (a, b) where $a : \alpha$ and $b : \beta$.
- Let α be a type in universe $\mathcal{U}$ and $\beta : \alpha \to \mathcal{U}$ be a family of type that for any $a : \alpha$,$\beta(a)$ is a type in $\mathcal{U}$. Then we can form the Π-type

$$\prod_{a:\alpha} \beta(a)$$

whose element is of the form $f : \prod_{a:\alpha} \beta(a)$ such that for any $x : \alpha$, $f(x) : \beta(x)$. Note that function type is actually an example of Π-type where β is a constant family of types. For this reason, we also call Π-types dependent functions. For example if $\mathrm{Vec}(\mathbb{R}, n)$ is the type of $\mathbb{R}^n$, then

$$n \mapsto \underbrace{(1, \cdots, 1)}_{n \text{ times}} : \prod_{m:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, m)$$

- We also have dependent cartesian product or Σ-type: Let α be a type in universe $\mathcal{U}$ and $\beta : \alpha \to \mathcal{U}$ be a family of types in $\mathcal{U}$, then the Σ-type

$$\sum_{a:\alpha} \beta(a)$$

whose element is of the form $(x, y) : \sum_{a:\alpha} \beta(a)$ such that $x : \alpha$ and $y : \beta(x)$. Similarly

$$\left(n, \underbrace{(1, \cdots, 1)}_{n \text{ times}}\right) : \sum_{m:\mathbb{N}} \mathrm{Vec}(\mathbb{R}, m)$$

2.1.1 Proposition as type

In type theory, a proposition p can be thought as a type whose elements is a proof of p.

Example 1. $1 + 1 = 2$ is a proposition. `rfl` is an element of type $1 + 1 = 2$ where `rfl` is the assertion that every term equals to itself.

Example 2. For two propositions p and q, the implication $p \implies q$ then can be interpreted as function $p \to q$. To say $\mathrm{imp} : p \to q$ is to say for any $\mathrm{hp} : p$ we have $\mathrm{imp}(\mathrm{hp}) : q$, or equivalently given any hp, a *proof* of proposition p, $\mathrm{imp}(\mathrm{hp})$ is a proof of proposition q.

Example 3. If $p : \alpha \to$ proposition $\forall x : \alpha, p(x)$ can be interpreted as a Π-type $\prod_{x:\alpha} p(x)$. To prove $\forall x : \alpha, p(x)$, we need to find an element of type $\prod_{x:\alpha} p(x)$, equivalently for any $x : \alpha$, we need to find an element of type $p(x)$, equivalently for any $x : \alpha$, we need to find a proof of $p(x)$.

Similarly, $\exists x : \alpha, p(x)$ can be interpreted as a Σ-type $\sum_{x:\alpha} p(x)$. To prove $\exists x : \alpha, p(x)$ is to find an element x of type α and prove $p(x)$, equivalently to find an element $x : \alpha$ and an element of type $p(x)$ and this is precisely $(x, p(x)) : \sum_{a:\alpha} p(a)$.

Theorems are true propositions, using the interpretation above, theorems are inhabited types and to prove a theorem is to find an element of the required type.

2.2 Lean and mathlib

mathlib is *the* collection of mathematical definition, theorems, lemmas built on Lean. mathlib includes topics in algebra, topology, manifolds and combinatorics etc. In this section, we are going to explain briefly how to use Lean with mathlib.

In Lean, new definition can be introduced with the following syntax:

```
def name (arg_1:type_1) ... (arg_n:type_n) : return_type := contents

def name' {arg_1:type_1} ... (arg_n:type_n) : return_type := contents
```

Sometimes return_type can be dropped when it can be inferred from contents. If an argument is surrounded by curly bracket instead of round bracket, then when the definition is invoked the said argument is implicit, i.e. name' a_2 ... a_n where a_i:$type_i$. To explicitly mention the said argument, one needs to use @name' a_1 ... a_n where a_i:$type_i$ Theorems or lemmas are introduced with the following syntax:

```
theorem name (arg_1:type_1) ... (arg_n:type_n) : content :=
begin
  -- proof of the theorem
end
```

To write a proof understandable to Lean, one need to use *tactic mode*. In Lean, one can use

- proof by induction: if the goal is a proposition about natural number n, induction n with n IH is to prove the proposition by induction. This command will change the current goal to two goals. The first goal is to prove the proposition for $n = 0$ and the second goal is to prove the proposition $n + 1$ with the additional inductive hypothesis IH;

```
theorem awesome_theorem_about_natural_number (n : ℕ) : proposition_n :=
begin
  induction n with n IH,

  a_proof_of_proposition_0

  -- (IH : proposition_n) is now in context
```

```
  a_proof_of_proposition_{n+1}
end
```

- proof by contradiction: if the goal is to prove proposition H, `by_contra absurdum` will add `absurdum :` $\neg H$ into the current context and turn the goal into proving `false`;

```
theorem awesome_theorem : awesome_proposition :=
begin
by_contra absurdum,

-- Now (absurdum : ¬ awesome_proposition) is in
 ↪  context and the goal is to prove falsehood.
a_proof_of_falsehood
end
```

- other tactics to finish or convert current goal into another set of goals:
  - `have H := content` will introduce a new proposition whose proof is given by `content`.

    `have H : some_proposition` will add one more goal of proving the proposition then introduce the proved proposition to the current context.
  - `unfold definition` is to unfold a definition to what is explicitly defined when the definition is introduced.
  - `simp` will simplify the goal with lemmas with an `@[simp]` tag. These lemmas are usually small and trivial like $\forall m \in \mathbb{N}, 0 + m = 0$[1].

    `simp only [h1, …hn]` is to simplify only using `h1` ... `hn`.
  - `rw` is for term rewriting. For example, if we have `h : lhs = rhs` or `lhs⟷rhs`, then `rw h` will replace every occurrence of `lhs` with `rhs` and `rw ←h` will replace every occurrence of `rhs` with `lhs`.

    `rw [h1, h2, ... , hn]` is the same as `rw h1, rw h2, ... , rw hn`.
  - Since `rw` and `simp` will change all occurrence, this sometimes would be inconvenient. `conv_lhs {tactics}` will confine the scope of `tactics` only to left hand side. Similarly `conv_rhs {tactics}` will confine the scope to right hand side.
  - Given (a proof of) proposition `H: h1 → h2`, then `apply H` will change the goal of proving `h2` to prove `h1`.
  - `ring` will try to prove the current goal using associativity and commutativity of addition and multiplication.

[1]this one is called `nat.zero_add`

- `linarith` is used when proving inequality from context. `linarith` is semi-automated, so it can work with inequalities with symbols or variables but only to a degree. If `linarith` failed, one has to either provide `linarith` with more propositions or use other tactics to change goal into something more manageable for `linarith`.

  `linarith [h1, ..., hn]` is equivalent to use `linarith` with additional (proofs of) propositions `h1` ... `hn`.
- If `H` is already in context then `replace H := content` will change `H` to a proof of the proposition that `content` is proving.

  `replace H : some_proposition` will add one more goal of proving `some_proposition` and then replace `H` to the proposition proven.
- `generalise H : lhs = var_name` will set `var_name` to `lhs` and add (proof of) the proposition `H : lhs = var_name` to the current context.
- `refl` (for reflexive) is used to prove proposition of the form `lhs = rhs` when `lhs` is **definitionally** equal to `rhs`. Definitional equality is more general than two string being literally identical but is less general than being (canonical) isomorphic. For example

$$\sum_{i=0}^{\infty} \frac{1}{2^i} = \sum_{j=0}^{\infty} \frac{1}{2^j}$$

  is a definitional equality but

$$\mathbb{R}^n = \text{Func}\left(\{0, \cdots, n-1\}, \mathbb{R}\right)$$

  is not a definitional equality (strictly speaking perhaps not an equality at all).
- `exact H` will prove current goal if the goal is definitionally equal to H.
- `suffices H : some_proposition` ask a proof of the current goal with additional `H`, then ask for a proof of `H`.
- `norm_cast` is convert the type of numbers. For example the current goal is $(x : \mathbb{R}) < (y : \mathbb{R})$ where x and y are of type $\mathbb{N}$, then after `norm_cast` the goal will become $x < y$. This should be simpler because $\mathbb{R}$ in `Lean` is equivalent classes of Cauchy sequence of $\mathbb{Q}$ while natural number is much easier to work with.

  `norm_num` is equivalent to `norm_cast, simp`.
- `ext` will convert the current goal with axioms of extensionality. For example if the goal is to prove equality of polynomial then after `ext` the goal would become to prove that every coefficient is equal; or if the goal is to prove equality of sets of type α $A = B$, then after `ext`, an

arbitrary element `x` of type α will be introduced to context then the goal will become to prove $x \in A \iff x \in B$. `ext var_name` will force `Lean` to introduce new variable under the identifier `var_name`.

  - If `H : ∃ x : type, property_about_x` is in the current context, `choose x hx using H` will introduce `x:type` with the assumption `property_about_x` to the current context.
  - If `H :` $p \wedge q$ is in the current context, then `H.1` is a proof of p and `H.2` is a proof of q.
  - If there is multiple goals, one can use `{ }` to focus on the first one.

A proposition if not atomic is either a conjunction, a disjunction, an implication, an equivalence, a negation or a proposition with universal quantifier or existential quantifier.

2.2.1 prove a conjunction

If goal is to prove a conjunction of the form $h_1 \wedge h_2$, `split` is used. It will change the current goal to two goals of proving h_1 and h_2 respectively. Then the general pattern is

```
theorem how_to_prove_conjunction (h_1 : Prop) (h_2 : Prop) : h_1 ∧ h_2 :=
begin
split,

proof_of_h_1

proof_of_h_2
end
```

2.2.2 prove a disjunction

If the goal is to prove a disjunction of the form $h_1 \vee h_2$, one can use `left` to change the goal to prove h_1 or `right` to change the goal to prove h_2. Let us assume h_1 is a true proposition :

```
theorem how_to_prove_disjunction (h_1 : Prop) (h_2 : Prop) : h_1 ∨ h_2 :=
begin
left,

proof_of_h_1
end
```

2.2.3 prove an implication

If the goal is to prove an implication of the form $p \implies q$, one can use `intro` hp to add hp:p a proof of p into the context and convert goal to prove q.

```
theorem how_to_prove_implication (p : Prop) (q : Prop) : p → q :=
begin
intro hp,

proof_of_q
end
```

If the goal is of the form $p_1 \to p_2 \to \ldots p_n$, one can use `intros` h$p_1$ …hp_n as an abbreviation of `intro` hp_1, `intro` hp_2,…, `intro` hp_n.

2.2.4 prove an equivalence

An equivalence of the form $p \iff q$ is by definition $p \implies q \wedge q \implies p$. Thus by `split` will change the goal to two goals, one to prove $p \implies q$, the other to prove $q \implies p$. Then use section 2.2.3.

2.2.5 prove a negation

A negation of the form $\neg p$ is by definition $p \implies \perp$. Thus `intro` hp will add hp:p to current context and convert the goal to prove a falsehood.

```
theorem how_to_prove_negation (p : Prop) : ¬p :=
begin
intro hp,

proof_of_falsehood
end
```

2.2.6 prove a proposition with ∀

A proposition of the form $\forall a : \alpha, p(a)$ where α is a type and $p : \alpha \to$ `Prop` can be proved also using `intro` x_0. This will add an arbitrary $x_0 : \alpha$ to the current context and change the goal to prove $p(x_0)$.

```
theorem how_to_proposition_with_universal_quantifier {α : Type} (p : α → Prop) : ∀ a : α, p a :=
begin
```

```
intro x_0,

a_proof_of_p(x_0)
end
```

If the goal is the form $\forall a_1 : \alpha_1, \forall a_2 : \alpha_2, \ldots, \forall a_n : \alpha_n, p\ a_1\ a_2\ \ldots\ a_n$ can be proved using `intros` a_1 a_2 … a_n as an abbreviation of `intro` a_1, `intro` a_2,…, `intro` a_n.

2.2.7 prove a proposition with ∃

A proposition of the form $\exists a : \alpha, p(a)$ where α is a type and $p : \alpha \to$ `Prop` can be proved by `use` x_0. This will convert the goal to prove $p(x_0)$.

```
theorem how_to_proposition_with_universal_quantifier {α : Type} (p : α → Prop) : ∃ a : α, p a :=
begin
a_construction_of_x_0

use x_0,

a_proof_of_p(x_0)
end
```

2.3 An example

To illustrate the above syntax and patterns, we present an example of defining mean and proving some basic properties thereof.

```
import data.real.basic
import tactic

noncomputable theory
open_locale classical

def mean (x y : ℝ) : ℝ := (x + y) / 2

theorem min_le_mean : ∀ x y : ℝ, min x y ≤ (mean x y) :=
begin
intros x y,
have ineq1 : min x y ≤ x := min_le_left x y,
```

```
13 have ineq2 : min x y ≤ y := min_le_right x y,
14
15 unfold mean, rw le_div_iff, rw mul_two,
16 apply add_le_add,
17 exact ineq1, exact ineq2,
18
19 linarith,
20 end
21
22 theorem mean_le_max : ∀ x y : ℝ, (mean x y) ≤ max x y :=
23 begin
24 intros x y,
25 have ineq1 : x ≤ max x y := le_max_left x y,
26 have ineq2 : y ≤ max x y := le_max_right x y,
27
28 unfold mean, rw div_le_iff, rw mul_two,
29 apply add_le_add,
30 exact ineq1, exact ineq2,
31
32 linarith,
33 end
34
35 theorem a_number_in_between :
36   ∀ x y : ℝ, x ≤ y → ∃ z : ℝ, x ≤ z ∧ z ≤ y :=
37 begin
38 intros x y hxy,
39 have ineq1 := min_le_mean x y,
40 have ineq2 := mean_le_max x y,
41 have min_eq_x := min_eq_left hxy,
42 have max_eq_y := max_eq_right hxy,
43 use mean x y,
44 split,
45
46 { conv_lhs {rw ←min_eq_x}, exact ineq1, },
47 { conv_rhs {rw ←max_eq_y}, exact ineq2, },
48 end
```

Line 1 will make basic properties of real available to use and line 2 will make all the tactics we discussed amongst other more advanced tactics available to use. We add line 4 so that `lean` would ignore the issue of computability and line 5 so that we can use proof by contradiction[2].

We define the mean value of two real numbers on line 7. Then `mean`[3] has

[2]`Lean` by default use constructivism where $\neg\neg p \implies p$ is not an axiom. Thus the law of excluded middle is not by default a tautology, hence one could not prove by contradiction.

[3]`mean` is not a function $\mathbb{R}^2 \to \mathbb{R}$ but a function $\mathbb{R} \to \mathrm{Func}(\mathbb{R}, \mathbb{R})$. This is called currying.

type $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$, `mean 1` has type $\mathbb{R} \to \mathbb{R}$ and `mean 1 2` has type $\mathbb{R}$.

We can introduce and prove theorems about `mean` that the mean value of two numbers is greater than or equal to the minimum of the two numbers but less than the maximum of the two numbers. This is from line 9 to line 33 where

- `min_le_left` is a proof of the proposition $\forall(x\ y : \alpha), \min(x, y) \leq x$ where α is an implicit argument with a linear order. In this case, `Lean` infers from context that α is $\mathbb{R}$. Thus `min_le_left x y` is a proof of `min x y ≤ x`.
- Similarly, `min_le_right` is a proof of the proposition $\forall(x\ y : \alpha), \min(x, y) \leq y$ where α is an implicit argument with a linear order. In this case, `min_le_right x y` is a proof of `min x y ≤ y`.
- Similarly, `le_max_left` is a proof of the proposition $\forall(x\ y : \alpha), x \leq \max(x, y)$ where α is an implicit argument with a linear order. In this case, `le_max_left` is a proof of `x ≤ max x y`.
- Similarly, `le_max_right` is a proof of the proposition $\forall(x\ y : \alpha), y \leq \max(x, y)$ where α is an implicit argument with a linear order. In this case, `le_max_right` is a proof of `y ≤ max x y`.
- `le_div_iff` is a proof that $0 < c \to (a \leq \frac{b}{c} \iff a \times c \leq b)$ where a, b, c are elements of a type with a linear ordered field structure. So by `rw le_div_iff`, the goal would change from `min x y ≤ (x + y) / 2` to `min x y * 2 ≤ x + y`. Since `le_div_iff` requires the assumption that $0 < c$, a new goal to prove that `0 < 2` is created after the original goal. This goal is proved by the final `linarith`.
- `div_le_iff` is proof that $0 < b \implies (\frac{a}{b} \leq c \iff a \leq c \times b)$ where a, b, c are elements of a type with a linear ordered field structure. So by `rw div_le_iff` the goal would change from `(x + y) / 2 ≤ max x y` to `x + y ≤ max x y * 2`. Since `div_le_iff` requires the assumption that $0 < b$, a new goal to prove $0 < 2$ is created after the original goal. This goal is proved by the final `linarith`.
- `mul_two` proves the lemma that $\forall n : \alpha, n \times 2 = n + n$ where α is a semiring. Thus `rw mul_two` would change the goal of proving `min x y * 2 ≤ x + y` (`x + y ≤ max x y * 2` resp.) to `min x y + min x y ≤ x + y` (`x + y ≤ max x y + max x y` resp.).
- `add_le_add` proves the lemma that $a \leq b \to c \leq d \to a + c \leq b + d$ where a, b, c and d are elements of an ordered additive commutative monoid. Since the goal now is to prove `min x y + min x y ≤ x + y`, by `apply add_le_add`, goal will be replaced by two goals of proving `min x y ≤ x` and `min x y ≤ y`. These are *exactly* `ineq1` and `ineq2`.

Chapter 3

Formalisation using `Lean`

Logistics of the formalisation

There are five main files in the formalisation where

1. `small_things.lean` formalised results about the trivial embedding of $\mathbb{Z}[X] \subset \mathbb{R}[X]$ and manipulation of inequality in real numbers common to all three parts;

2. 1234

3.1 Countability argument

3.2 Liouville's theorem and Liouville's number

3.3 Hermite's proof of transcendence of e

Reflection

The original motivation behind this formalisation project is not exactly to formalise some basic theorems in transcendental number theory for the sake of formalisation.

Bibliography

[AMK15] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem proving in Lean*. 2015.

[Bak90] Alan Baker. *Transcendental number theory*. Cambridge university press, 1990.

[BCM20] Kevin Buzzard, Johan Commelin, and Patrick Massot. "Formalising perfectoid spaces". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2020, pp. 299–312.

[Bou98] Nicolas Bourbaki. *Elements of the History of Mathematics*. Springer Science & Business Media, 1998.

[Boy+94] Robert Boyer et al. "The QED manifesto". In: *Automated Deduction–CADE* 12 (1994), pp. 238–251.

[Can32] G Cantor. *Uber eine Eigenschaft des lnbegriffes aller reelen algebraischen Zahlen. Crelles J. 77 (1874), 258-262; also in Gesammelte A bhandlungen*. 1932.

[Can78] Georg Cantor. "Ein beitrag zur mannigfaltigkeitslehre". In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* 1878.84 (1878), pp. 242–258.

[Gel34] Aleksandr Gelfond. "Sur le septieme probleme de Hilbert". In: *Известия Российской академии наук. Серия математическая* 4 (1934), pp. 623–634.

[Gon+13] Georges Gonthier et al. "A machine-checked proof of the odd order theorem". In: *International Conference on Interactive Theorem Proving*. Springer. 2013, pp. 163–179.

[Gon08] Georges Gonthier. "Formal proof–the four-color theorem". In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.

[Hal+17] Thomas Hales et al. "A formal proof of the Kepler conjecture". In: *Forum of mathematics, Pi*. Vol. 5. Cambridge University Press. 2017.

[Hal98] Thomas C Hales. "The kepler conjecture". In: *arXiv preprint math.MG/9811078* (1998).

[HUW14] John Harrison, Josef Urban, and Freek Wiedijk. "History of Interactive Theorem Proving." In: *Computational Logic*. Vol. 9. 2014, pp. 135–214.

[Kem16] Aubrey J. Kempner. "On Transcendental Numbers". In: *Transactions of the American Mathematical Society* 17.4 (1916), pp. 476–482. ISSN: 00029947. URL: `http://www.jstor.org/stable/1988833`.

[KK11] Juliette Kennedy and Roman Kossak. *Set Theory, Arithmetic, and Foundations of Mathematics: Theorems, Philosophies*. Vol. 36. Cambridge University Press, 2011.

[Lam04] M Lambert. "Mémoire sur quelques propriétés remarquables des quantités transcendentes circulaires et logarithmiques". In: *Pi: A Source Book*. Springer, 2004, pp. 129–140.

[Lan66] Serge Lang. *Introduction to transcendental numbers*. Addison-Wesley Pub. Co., 1966.