Euclidean Domains in Lean

Louis Carlin

February 28, 2018

Introduction

In this report I detail the formalisation of Euclidean Domains in an interactive theorem proving language Lean. I start with a brief explanation of Lean and some of its features. I then give an overview of my work on Euclidean domains, with a particular focus on the Euclidean Algorithm and the role well-founded relations play in proving its properties. My finished work can be found at `https://github.com/semorrison/2017-summer-students/blob/master/src/Louis/euclidean_domain.lean`.

1 An overview of Lean

1.1 A quick look at type theory

Lean is based on type-theoretic system, rather than the set theory traditionally used by most mathematicians. Initially, working with type theory is not too different to working with sets. We write $x : \alpha$ to mean "x is something of type α" rather than $x \in \alpha$ to mean "x is an element of α". We can also reason about many of the sets we are familiar with such as $\mathbb{N}$ or $\mathbb{Z}$ as types.

However, in type theory objects belong to a single type and are always identified as being of that type. It is not possible to talk about an object x in Lean without giving it a type. Nor is it really possible to talk about something as being an object of more than one type. So for example $5 : \mathbb{N}$ and $5 : \mathbb{Z}$ are distinct objects. Luckily lean has a fairly versatile coercion system which allows us to convert between objects of different types, although the process can be clunky at times.

We can build types out of other types, so for example $\alpha \to \beta$ is the type of functions from α to β. Lean's axiomatic system also extends simple type theory by implementing Types themselves as objects, meaning they themselves must have types. So, for example $\mathbb{N}$: `Type` and `Type : Type 1`. In fact, in general `Type` u `:` `Type` $(u+1)$. This infinite type hierarchy is Lean's way of avoiding the contradictions brought about by "the type of all types".

The main advantage of type theory in the context of Lean is that it is easier to model computationally. When we have some object represented as a bunch of bits it makes sense to give it a type so that we can interpret those bits. The propositions as types paradigm explained below also means type theory feels like a tidier axiomatic system than traditional set theory.

1.2 Propositions as types

Lean treats propositions as types. A proposition `p : Prop` is the type of proofs of `p`. So when we say `h : p`, we mean that `h` is a proof of `p`. For example `h : 5 > 3` is a proof that 5 is greater than 3. To prove `p`, it suffices to show we have something of type `p` and thus if our proofs type-check correctly then they are valid[1].

Under this paradigm of propositions as types, a proof that `p : Prop` implies `q : Prop` is the same as function that takes a proof of `p` and gives a proof of `q`. This means that a proof of `p ⟹ q` is something of type `p → q`, and thus we can write `p → q` in place of `p ⟹ q`. One other thing to note is that under the principle of "proof irrelevance" Lean treats all proofs of a proposition `p` as definitionally equal, so any two proofs of `p` are in effect the same.

1.3 Lean Syntax

The most important notation to understand in Lean is function definition. The following example is the "if then else" function as it appears in Lean. The function takes a decidable proposition `c` as input as well as return values `t e :` α. If `c` is true then it returns `t`, otherwise it returns `e`.

```
def ite (c : Prop) [h : decidable c] {α : Sort u} (t e : α) : α
:= decidable.rec_on h (λ hnc, e) (λ hc, t)
```

The `def` keyword indicates the start of a definition and is followed by the name of the function `ite`. The parentheses `(c : Prop)` indicate that the function takes a proposition `c : Prop` as an explicit argument. The square brackets indicate to Lean that what is between them is an input that Lean should be able to find for itself with its type class resolution system. In this case `decidable c` is an inductive type which either holds a proof that `c` is true or that it isn't.

```
class inductive decidable (p : Prop)
| is_false (h : ¬p) : decidable
| is_true  (h : p) : decidable
```

If Lean can't find an instance `h : decidable c` for a given input `c` then it will give an error message telling you type class resolution has failed.

Going back to the definition of `ite`, the curly brackets {α `: Sort u`} indicate that the argument inside them is implicit, which is to say it can be figured out from later arguments. In this case α is the type of the final two arguments `t e :` α, so Lean is able to work out what you mean from the types of `t` and `e`. The type after the colon is the return type of the function (in this case α since `ite` will return `t` or `e`). Finally, the part after the `:=` is the definition of the function. In this case it makes use of a recursion principle `rec_on` which Lean defines automatically for inductive types. The notation λ `hc, t` is how we write a function which takes an argument `hc` and returns the value `t`. While normally we might have to write, λ `hc : c, t :` α, Lean has enough information in this case to figure out these types itself so we can omit them.

[1] This is assuming there are no bugs in the Lean checker. The Lean FAQ gives more details about the soundness of the kernel.

exampletactic.png

Figure 1: An example of a proof in tactics mode.

tactic.png

Figure 2: An example context display from where the cursor is in Figure 1. The `intro` tactic has been used to introduce the hypothesis `hpq : p ∧ q`.

1.4 Tactics

Writing out long proofs consisting entirely of anonymous λ functions is time-consuming and finicky. Lean does define some syntactic sugar to make proofs read more like traditional mathematical proofs. For example, we can write `assume hp : p,` instead of λ `hp : p`, and `lemma` and `theorem` are synonyms for `def`. However, the real strength of Lean in proving things is in tactics mode.

A tactic is a monad which we use to automate repetitive tasks in proving things by taking the current state ('context') of what we know and what we are aiming to prove, and performing some deductive step with it. One of the unique features that Lean offers is that tactics are written in Lean itself rather than an external meta-language.

We tell Lean to enter tactics mode with a `begin end` block. Once in tactics mode Lean gives us a printout of the current context, which is what we know at that point in the proof. One can move the/cursor to different points and the printout will update with the context at that specific moment. The turnstile symbol $\vdash$ indicates what we are trying to show.

There are several tactics which feature commonly in the proofs I wrote. The `split` tactic turns a goal of the form `p ∧ q` to two goals `p` and `q`. `intros` introduces any hypotheses needed to proved our goal, so if we had a goal of `p → q → r`, then intros would give us hypotheses `h_1 : p` and `h_2 : q` and change the goal to `r`. Writing `rw h` where `h` is some proof of an equality allows us to rewrite by substituting for the right side of the equality whenever the left side appears in the goal. `simp` tries to simplify the goal using predefined simplification lemmas which have been marked for its use. All these tactics can also be used on a hypothesis `h` rather than the goal by writing `at h` after invoking them.

1.5 Constructivism

Lean encourages a constructive approach to mathematics. This means that proofs usually give an explicit instance of the thing you are trying to prove and techniques like proof by contradiction are discouraged. Lean does allow for the non-constructive axiom of choice, stating it as follows[2]:

```
/- the axiom -/
axiom choice {α : Sort u} : nonempty α → α
```

However, it must be imported from a file 'classical.lean' in Lean's library, and proofs or functions which rely upon it must be marked as non-computable.

The law of the excluded middle follows from the axiom of choice in Lean's axiomatic system[3] and is defined amongst other non-constructive principles as a theorem in 'classical.lean'.

While trying to avoid using the axiom of choice and law of the excluded middle was initially unintuitive to me, it has the substantial advantage that anything you write in Lean without them is computable. So for example my implementation of the Euclidean Algorithm is not just an abstract proof that such an algorithm exists, but can actually be executed in arbitrary Euclidean Domains.

2 Euclidean domains

A euclidean domain α is an integral domain with a divison algorithm, such that for any two elements $a, b : \alpha$, there exists $q, r : \alpha$ such that $a = qb + r$. We require that r is "smaller" than the thing we are dividing by by defining Euclidean Domains to have a valuation function $\text{val} : \alpha \to \mathbb{N}$ such that $\text{val}\ r < \text{val}\ b$.

2.1 Defining Euclidean Domains in Lean

Lean's typeclass system allows us to define new structures using the `extend` keyword as pre-existing structures with additional requirements. Integral domains are already defined in Lean as an extension of commutative rings, so I defined Euclidean domains as an extension of them.

```
class euclidean_domain (α : Type) extends integral_domain α :=
( quotient : α → α → α )
( remainder : α → α → α )
( witness : ∀ a b, (quotient a b) * b + (remainder a b) = a )
( valuation : euclidean_valuation remainder)
```

The first three fields of the `euclidean_domain` class are the quotient and remainder functions (sometimes denoted `a/b` and `a%b` respectively) as well as a

[2]Lean goes on to prove the more traditional set-theoretic statement of the axiom of choice using this axiom

[3]see https://en.wikipedia.org/wiki/Diaconescu%27s_theorem

proof of the fact that the q and r these give us actually do satisfy $a = bq + r$. The fourth field refers to a `euclidean_valuation` which is defined as so:

```
definition euclidean_valuation {α} [has_zero α] (r : α → α → α)
      := { f : α → ℕ // ∀ a b, b = 0 ∨  (f(r a b)) < (f b)}
```

The notation `{ f // g f}` represents a dependent pair type, where the type of the second element depends on the first element. In this case we use it to define a `euclidean_valuation` as the pair consisting of a function to the natural numbers as well as a proof that `b = 0 ∨ f(r a b) < f b`, where in this case r will be our remainder function.

There were a couple of key choices I had to make in the definition of `euclidean_domain`. Firstly note that we require an explicit `quotient` and `remainder` function, rather than just the fact that they exist or that it is in some way possible to write $a = bq + r$ for any a, b. The motivation for this is that given Lean's computational nature we want to be able to actually compute things (especially Euclid's algorithm) in arbitrary euclidean domains.

The valuation function (and its proof) are also explicitly defined. This is somewhat unusual as the traditional definition just requires that a euclidean domain can be equipped with *some* valuation function with the desired property. I initially defined it as follows:

```
(valuation : ∃ f : α → ℕ,∀ a b : α,  b = 0 ∨ f (remainder a b)
    < f b)
```

The problem with this approach was that Lean didn't like me using the non-constructive axiom of choice to define a non-computable well-founded relation[4] and thus I couldn't prove the Euclidean Algorithm was a well-founded recursion. The main disadvantage to the way I ended up doing it with an explicit valuation function is that we can no longer expect two euclidean domains of the same type with the same `quotient` and `remainder` functions to be equal (unless they happen to have the same valuation), and must instead reason about them as isomorphic.

Finally, some definitions of euclidean domains require valuation functions to have the additional property that for all a, b the valuation of a is less than or equal to the valuation of ba. Initially I excluded this from my definition because I read[5] that given a valuation function $g : \alpha \to \mathbb{N}$, it is possible to define a valuation $f : \alpha \to \mathbb{N}$ with this second property as $f(a) = \min_{b:\alpha} g(ba)$. However, the problem with this valuation that we can only guarantee it acts as a valuation for *some* divison algorithm. It is not necessarily defined in terms of the same `quotient` and `remainder` functions as our original valuation function.

There are several results which match our intuition from integer division that this additional property allows us to prove, but that are not necessarily true without it. For example, that $0/a = 0$ for any non-zero a.

The way I would approach fixing this is to define euclidean domains with this property as an extension of normal euclidean domains as well as define a function

[4] explained below

[5] https://www.jstor.org/stable/2316324

from a normal euclidean domains to (noncomputable) euclidean domains with this property using the valuation function one can construct.

3 The Euclidean Algorithm

The Euclidean Algorithm is one of the main motivations of the definition of Euclidean Domains. It takes any two elements a, b of a Euclidean Domain R and gives a "greatest common divisor" of a and b.

An element $d : R$ is a greatest common divisor of a and b if $d \mid a$, $d \mid b$ and $\forall x : R, x \mid a \wedge x \mid b \implies x \mid d$. We commonly write $\gcd(a, b)$ to denote a particular greatest common divisor of a and b. However it is important to note that greatest common divisors are not necessarily unique. In fact they are never unique in our definition of a Euclidean domain: if d is a gcd of a a and b in a ring, then so is its additive inverse $-d$.

The Euclidean Algorithm works by calculating a sequence $r_0, r_1, \ldots, r_k$, where $r_0 = a$, $r_1 = b$ and for $n > 1$, $r_n = r_{n-2} \ \% \ r_{n-1}$. The sequence terminates when we reach r_k such that $r_k = 0$ and the gcd is r_{k-1}. The proof that this is actually the gcd goes by induction. If something divides r_n and r_{n-1} then we can show it divides r_{n-2} and if something that divides a and b must divide r_{n-2} and r_{n-1} then we can show it must divide r_n.

One thing to note is that the Euclidean Algorithm is not actually decidable in the definition of `euclidean_domain` which I gave in 2.1. This is because Lean does not assume equality is decidable by default, and so it cannot actually tell when the sequence of r_n ends since it cannot tell when something equals zero. I fixed this by extending `euclidean_domain` in `decidable_euclidean_domain`:

```
class decidable_euclidean_domain (α : Type) extends
    euclidean_domain α:=
(decidable_eq_zero : ∀ a : α, decidable (a = 0))
```

3.1 The Extended Euclidean Algorithm

The Extended Euclidean Algorithm adds to the Euclidean Algorithm by also returning two elements $x \ y : \alpha$ (called Bézout coefficients) such that $ax + by = \gcd(a, b)$. It does this by defining two additional sequences s_n and t_n. We define $s_0 = 1$, $s_1 = 0$, and $s_n = s_{n-2} - (r_n / r_{n-1}) s_{n-1}$ for $n > 1$. Similarly we define $t_0 = 0$, $t_1 = 1$, and $t_n = t_{n-2} - (r_n / r_{n-1}) t_{n-1}$ for $n > 1$. We show that these are the desired coefficients by using induction to show that for all n, $r_n = as_n + bt_n$.

3.2 Well-founded recursion

Whenever we define a recursive function Lean requires us to show that it is actually well-defined. Since the Euclidean Algorithm is defined recursively this was a necessary step. The method Lean provides for us to do this is demonstrating that the inputs used in the recursive call are smaller than current inputs according to some well-founded relation.

To understand what this means one first needs to understand what a well-founded relation is. Lean defines a relation `r :` $\alpha \to \alpha \to \alpha$ as well-founded when all the elements of α are 'accessible':

```
inductive well_founded {α : Sort u} (r : α → α → Prop) : Prop
| intro (h : ∀ a, acc r a) : well_founded
```

For an element to be accessible, all the other elements less than it (by the relation `r`) must be accessible:

```
inductive acc {α : Sort u} (r : α → α → Prop) : α → Prop
| intro (x : α) (h : ∀ y, r y x → acc y) : acc x
```

How one can think of this is that there must be some 'base' elements for which nothing is less than, and which are therefore accessible. We then build upwards from these foundations because the things which are only greater than them are therefore also accessible, and so on. It is important to note that there can be no infinite chains of less-than relations in a well-founded relation since we cannot show something is accessible without showing that if we descend down the tree of less-than relations far enough we will encounter only things with nothing less than them. We cannot have any cycles for a similar reason.

Conveniently, Lean can use its typeclass resolution system to show basic recursive functions decrease on a well-founded relation all by itself. For example this factorial function works fine as is:

```
def fac : ℕ → ℕ
| 0 := 1
| (n + 1)  := fac n * (n+1)
```

However, it needs a bit of help for more complex recursive functions. This definition of natural number division[6] requires us to demonstrate that the first argument `x - y` given in the recursive call is smaller than the current first argument `x`:

```
def div’ : ℕ → ℕ → ℕ
| x y :=
  if h : 0 < y ∧ y ≤ x then
    have x - y < x,
      from sub_lt (lt_of_lt_of_le h.left h.right) h.left,
    div’ (x-y) y + 1
  else 0
```

If we remove the proof that `x - y < x` then Lean will give us the error `failed to prove recursive application is decreasing`.

In addition to their use in showing recursive functions 'terminate', well-founded types allow us to perform well-founded induction, and prove some predicate P is true for all objects of a well-founded type α by proving $\forall x : \alpha, (\forall y : \alpha, y < x \to Py) \to Px$.

[6]From Theorem Proving in Lean

3.3 The First implementation

In defining the Euclidean Algorithm in Lean it was important to not only have it return the gcd as expected, but also to have a proof that what it returned *was* the gcd. My first implementation of the Euclidean Algorithm satisfied this requirement by taking proofs as part of its input and returning a value paired with the fact that that value was actually a gcd. This is how I defined the input and output types:

```
structure common_divisor {α : Type} [R: comm_ring α] (a b : α) :=
(value : α)
(divides_a : value ∣ a)
(divides_b : value ∣ b)

structure greatest_common_divisor {α : Type} [R: comm_ring α] (a
    b : α) extends common_divisor a b :=
(greatest : ∀ d : common_divisor a b, d.value ∣ value)

/- this is the return type -/
structure bezout_identity {α : Type} [R: comm_ring α] (a b : α):=
(x y : α) -- coefficients
(gcd : greatest_common_divisor a b)
(bezout : gcd.value = a * x + b * y)

/- this is the input type -/
structure eea_input {α : Type} (a b : α) [euclidean_domain α] :=
(rp rc xp xc yp yc: α)
(bezout_prev : rp = a * xp + b * yp)
(bezout_curr : rc = a * xc + b * yc)
(divides : ∀ x : α, ∣xrp ∧ ∣xrc → ∣xa ∧ ∣xb)
(greatest_divisor : ∀ d : common_divisor a b, d.value ∣ rp ∧ d.
    value ∣ rc)
```

This approach wasn't particularly good. The actual statement of the algorithm ended up being almost two pages long because I had to write the functions transforming the proofs I took as input to the proofs I needed to give as output. This meant that when I ran into issues (primarily trying to use non-computable valuations to show it was well-founded recursion) it was difficult to work out exactly where the problems were. It would also have been much less intuitive to have to work with the structure `bezout_identity` I defined than to be simply given a gcd and only given the proofs of its properties if you asked for them.

3.4 The Second implementation

My second implementation was much simpler. I modelled it off the gcd function for natural numbers which already existed in Lean's mathlib library. The function does not take or give any proofs and is defined in only four lines. Note the use of the lemma `gcd_decreasing` to show that the first argument is decreasing.

```
def gcd {α : Type} [ed : decidable_euclidean_domain α] :
    α → α → α
| x y := if x_zero : x = 0 then y
         else have h : has_well_founded.r (y % x) x :=
    gcd_decreasing x y x_zero,
              gcd (y%x) x
```

We prove the properties of this function externally. The advantage of doing this is that it is a much more modular approach. We can prove each property individually and if there are problems it will be much more apparent which part of the code they appear in. In addition, if we want to prove further properties of the function we don't have to alter its return type as we can just write a new lemma.

The method[7] I used to prove most of the properties of gcd was to prove a new induction principle as follows:

```
theorem gcd.induction {α : Type} [ed: decidable_euclidean_domain
    α]
                {P : α → α → Prop}
                (m n : α)
                (H0 : ∀ x, P 0 x)
                (H1 : ∀ m n, m ≠ 0 → P (n%m) m → P m n) :
            P m n :=
@well_founded.induction _ _ (has_well_founded.wf α) (λm, ∀n, P m
    n) m (λk IH,
    by {cases decidable.em (k=0), rw h, exact H0,
        exact λ n, H1 k n (h) (IH (n%k) (neq_zero_lt_mod_lt n k h
    ) k)}) n
```

This code is somewhat cryptic, but the basic idea is we prove a less powerful but more specific induction principle from the principle of well-founded induction. We can think of `H0 : ∀ x, P 0 x` as the base case, `H1 : ∀ m n, m ≠ 0 → P (n%m) m → P m n` as the inductive step, and the return type `P m n` as what our new induction principle lets us conclude given these two things.

With this induction principle the proof that our algorithm returns something which divides a and b is as follows:

```
theorem gcd_dvd {α : Type} [ed: decidable_euclidean_domain α] (a
    b : α) : (gcd a b ∣ a) ∧ (gcd a b ∣ b) :=
gcd.induction a b
  (λ b, by simp)
  (λ a b aneq,
  begin
  intro h_dvd,
    rw gcd, simp [aneq],
```

[7] based on a similar method for natural number gcds in mathlib

```
    cases h_dvd,
    split,
    {exact h_dvd_right},
    {
      conv {for b [2] {rw ←(euclidean_domain.witness b a)}},
      have h_dvd_right_a:= dvd_mul_of_dvd_right h_dvd_right (b/a),

      exact dvd_add h_dvd_right_a h_dvd_left
    }
  end )
```

The first lambda function is a proof of the base case using the handy tactic `simp` which simplifies the goal using pre-defined lemmas which have been marked for it to use. In this case it simplifies using the fact that `gcd(0,b) = b`, then knows that `b | 0` and `b | b`.

The second lambda function proves the inductive step by substituting the definition of `gcd` in, simplifying with `aneq : a ≠ 0`, then showing that if something divides `b%a` and `a` then it also divides `b`. The `conv` tactic allows us to apply the tactic we give it – in this case `rw` – to a specific term in the goal. We use it here to rewrite a single specific term and not the other instances of said term in the goal.

We show that the `gcd` is the 'greatest' divisor as follows:

```
theorem dvd_gcd {α : Type} [ed: decidable_euclidean_domain α] {a
    b c : α} : c ∣ a → c ∣ b → c ∣ gcd a b :=
gcd.induction a b
  (λ b,
  begin
    intros dvd_0 dvd_b,
    simp, exact dvd_b
  end)
  (λ a b hna,
  begin
    intros d dvd_a dvd_b,
    rw gcd, simp [hna],
    exact d (dvd_mod dvd_b dvd_a) dvd_a,
  end)
```

My implementation of the Extended Euclidean Algorithm was extremly similar to the pre-existing one for natural numbers in Lean. It is called by `xgcd` and uses an auxilary inner function as `xgcd_aux` to perform the actual recursion.

```
def xgcd_aux {α} [decidable_euclidean_domain α] : α → α → α →
    α → α → α → α × α × α
| r s t r' s' t' := if r_zero : r = 0 then (r', s', t')
    else have has_well_founded.r (r' % r) r, from
    neq_zero_lt_mod_lt r' r r_zero,
    let q := r' / r in xgcd_aux (r' % r) (s' - q * s) (t' - q * t
    ) r s t

def xgcd {α} [decidable_euclidean_domain α] (x y : α) : α × α :=
     (xgcd_aux x 1 0 y 0 1).2
```

I went on to prove the properties we desire its output to have with similar proofs to the previous ones using `gcd.induction`.