Bachelor Computer Science

Bachelor thesis

A Formally Verified Proof of the Mason-Stothers Theorem in Lean

by

Jens Wagemaker

August 1, 2018

Supervisor: Dr. Jasmin Christian Blanchette

Daily supervisor: Dr. Johannes Hölzl

Second examiner: Dr. Sander Dahmen

a + b + c = 0

Department of Computer Science
Faculty of Sciences

VU UNIVERSITY AMSTERDAM

Abstract

Although having computer checked proofs has several advantages, the use of interactive proof assistants has not yet spread into the mathematical community. Problems that withhold mathematicians from using proof assistants include: lack of automation, libraries and expertise. Lean is a new proof assistant, developed towards use by mathematicians. I used Lean to formalize the Mason-Stothers theorem, which is a number theoretical result whose proof is short and yet contains many algebraic concepts. To develop the proof I created a reusable formal library including results on polynomials and unique factorization domains.

Title: A Formally Verified Proof of the Mason-Stothers Theorem in Lean
Author: Jens Wagemaker, Jens.Wagemaker@gmail.com
Supervisor: Dr. Jasmin Christian Blanchette
Daily supervisor: Dr. Johannes Hölzl
Second examiner: Dr. Sander Dahmen
Date: August 1, 2018

Department of Computer Science
VU University Amsterdam
de Boelelaan 1081, 1081 HV Amsterdam
http://www.cs.vu.nl/

Contents

Acknowledgement

I want to thank Johannes Hölzl. He was my daily supervisor. Through his help I was able to understand and use the Lean theorem prover. In the formalization he helped me with several proofs. He had the patience to discuss the definitions I introduced in the formalization. And his initial write-up on polynomials formed the starting point of the formalization. He reviewed and helped structuring this thesis. I want to thank Jasmin Blanchette for initiating and introducing me to this project. He reviewed my writings, and guided me with his knowledge on writing. I want to thank Sander Dahmen for sharing his knowledge as an experienced mathematician and for suggesting Mason-Stothers. Moreover, I want to thank Robert Lewis for his advice.

1. Introduction

Mathematical proofs are often hard to check for correctness. The proof presented by Mochizuki, who claims to have solved the *abc*-conjecture, is a prime example (Castelvecchi, 2015). His proof, and the background he developed for it, is so vast and new that it is taking years, and will continue to take years, to be understood and peer-reviewed by the mathematical community. As of this moment there is no consensus on the correctness of his proof.

We can use computers to verify the correctness of proofs. The process of creating proofs that are so precise and complete that a computer can verify their correctness is called mathematical formalization. In practice, one interacts with a proof assistant, which in turn generates a formal proof, which is then checked by the proof checker that is part of the proof assistant.

Having computer verified proofs has several advantages. In the first place, having a proof that is verified by a computer gives a high level of trust in the correctness, as we only have to trust the proof checker (including its logic and the system on which it runs) and the translation of informal (textbook style) statements to formal statements. Second, formalization helps organizing mathematics. Third, formalizing reveals lower level mathematics, which is mathematics that is currently not explicitly formulated. Fourth, formalization could be used for education in various ways. My first contact with a proof assistant was in a logics course, where the homework assignments had to be made in a proof assistant that provided instant feedback.

However, over 99% of the mathematical community is not using proof assistants, because the current systems are too laborious to use. Sometimes, one line of a proof can take a day to formalize. Obstacles that make proof assistants laborious to use are the lack of automation, libraries and expertise. In order for a mathematician to formalize anything, he or she needs to know the proof assistant and its libraries. A mathematician does not want to spend time on trivial lemmas, hence automation is needed. Nor does a mathematician want to develop the undergraduate (or graduate) mathematics that is needed before he or she can formalize his or her new results.

Lean is a new proof assistant that is co-designed by a mathematician, with the needs of working mathematicians foremost in mind. It attempts to combine the strength of the two leading proof assistants (de Moura et al., 2015). Lean's logical foundation is a dependent type theory that is similar to the foundations of the Coq proof assistant. Being roughly equally expressive as Coq, it aims to improve on Coq by having a small kernel, and having a framework that is designed towards automation. From the Isabelle proof assistant, Lean draws inspiration for the construction of its mathematical libraries, as it uses the type class mechanism found in Isabelle/HOL. It aims to even Isabelle in automation, while having the merit of a more expressive logic.

In this bachelor thesis I used the Lean proof assistant to formalize the Mason-Stothers theorem (Mason, 1984; Stothers, 1981) and a proof (Snyder, 2000). This number theoretical result is the polynomial analogue of the *abc*-conjecture. It has a short proof and yet contains many algebraic concepts, which make it an excellent test case for formalizing number theory using the Lean proof assistant. To develop the proof I created a reusable formal library including results on polynomials and unique factorization domains. The formalization was completed up to a single gap in the mathematical preliminary.

This thesis is divided in 7 chapters. Chapter 2 describes the Mason-Stothers theorem and the mathematical machinery that it uses. Chapter 3 introduces the subject of mathematical formalization and the Lean proof assistant. Chapter 4 describes the formalization of the reusable library and the Mason-Stothers theorem. Chapter 5 presents a comparison between my Lean formalization and a similar development in the Isabelle proof assistant. Chapter 6 presents a discussion of the usability of Lean for mathematicians and future work. Finally, Chapter 7 draws conclusions.

2. The Mason-Stothers Theorem

2.1. Mathematical Preliminaries

Rings Rings are sets which have two binary operations, the addition ($+$), and the multiplication ($\times$), defined on them. Both $+$ and $\times$ are associative, and the multiplication distributes over addition. The precise definition of a ring is given in Appendix A. Rings are a generalization of the integers ($\mathbb{Z}$) and polynomials over $\mathbb{Z}$. There are many types of rings. In particular we can form the following hierarchy of different types of rings: fields $\subset$ Euclidean domains $\subset$ principal ideal domains $\subset$ unique factorization domains $\subset$ integral domains $\subset$ rings. For proving the Mason-Stothers theorem it is sufficient to have fields, unique factorization domains and integral domains at our disposal. In this thesis I denote an arbitrary ring by R, unless specified otherwise. And I always assume that a ring has a multiplicative *identity*; this is in accordance with the definition of rings in Lean.

Integral domains Integral domains are commutative rings with $0 \neq 1$, which in addition do not have zero-divisors, i.e. we have $a \times b = 0$ implies $a = 0$ or $b = 0$ for all a, b in R. Integral domains are at the near bottom in the above hierarchy.

Unique factorization domains Unique factorization domains (UFDs) generalize the unique prime factorization property of the integers ($\mathbb{Z}$). However, in a general UFD the factorization into *irreducible* factors is only unique up to order and *associates*. Both 'irreducible' and 'association' are related to *units*. A unit is an element that has a multiplicative inverse. A non-zero non-unit element is irreducible if it cannot be factored into two non-units. Two elements are said to be associated if they are equal up to multiplication by a unit. In a UFD we have certain division properties. An element a *divides* an element b, notation $a|b$, if there exists an element c such that $b = a * c$. In a UFD any two elements have a *greatest common divisor* (gcd). Two elements are said to be *coprime* if their gcd is a *unit*.

In Section 4.1 I give precise definitions of UFDs and related notions. For units, irreducible, the associated relation, UFDs, the divides relation, gcds, and coprime see Definitions 4.1.1 to 4.1.7.

Fields A field is a commutative ring with $0 \neq 1$ in which every element that is not equal to zero is a unit. Fields are at the top in the above hierarchy. For any ring R there is a unique map from the integers ($\mathbb{Z}$) to R that is compatible with the addition and multiplication (i.e. a unital ring homomorphism). The smallest integer $n > 0$ which

gets mapped to 0 under this map is called the *characteristic* of the field. If no such n exists the field is said to have characteristic 0.

Polynomials Polynomials are expressions of the form $a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$, with $a_i \in R$ for some ring R. These expressions can be interpreted in two ways, as *polynomial functions* or as *polynomial forms*.

Polynomial function Here we see polynomials as functions from R to R, where we use the addition and multiplication from R to interpret $+$ and $\times$, and interpret X as a variable.

Polynomial form We give no meaning to the symbols, and treat it as a formal expressions. In this case the expression above simply represents an ordered list of finitely many elements of R (the a_i), i.e. $(a_0, a_1, a_2, \ldots, a_n)$, where the last element (a_n) is non-zero.

That the above two interpretations are not equal becomes clear when we consider X and X^2 as polynomials over the finite field $\mathbb{F}_2$, where they are the same as polynomial functions, but not the same as polynomial forms. In this thesis the term polynomial is reserved for polynomial forms.

The set of polynomials over a ring R, denoted by $R[X]$, has a ring structure as well. Depending on the type of the ring R the polynomial ring has additional features. A polynomial ring over an integral domain is an integral domain. A polynomial ring over a UFD is a UFD. And a polynomial ring over a field is a Euclidean domain (and therefore also a UFD by the aforementioned algebraic hierarchy). For polynomials a *formal derivative* can be defined, which we denote by f'. In the above polynomial expression the term a_nX^n is called the *leading term*, and a_n is the *leading coefficient*. If $a_n \neq 0$, we call n the *degree* of the polynomial. The degree of the zero polynomial is defined as $\deg(0) = 0$ in this thesis and in Lean. A polynomial is called *monic* if the leading coefficient equals 1. A polynomial of the form aX^0 is said to be *constant*.

In Section 4.2 I give precise definitions of polynomials and related notions. For polynomials over an arbitrary ring, *monomials*, polynomial addition, polynomials multiplication, polynomial features (e.g. degree and leading coefficient), and formal derivatives see Definitions 4.2.1 to 4.2.6.

Polynomial normal form When we have polynomials over a field K, any polynomial $f \in K[X]$ can be written in the normal form $f = c\,p_1p_2 \ldots p_n$, where c is a constant polynomial, and the p_i's are all monic and irreducible. In the normal form we decompose f into a constant factor and a multiset of monic irreducible factors. In the case that $f = 0$, we set the multiset of monic irreducible factors equal to the empty set, which then gives a unique normal form.

The radical of a polynomial The radical of a polynomial f, denoted by $\mathrm{rad}(f)$ is the product of all distinct monic irreducible factors dividing f. I.e. if we write f in an

adapted normal form, were we group equal factors of p_i into factors of the form $p_i^{a_i}$, $f = c\,p_1^{a_1}p_2^{a_2}\dots p_n^{a_n}$, with c a constant polynomial, and the p_i's monic and irreducible, the $a_i > 0$ for all i, and in addition we assume that the p_i's are distinct, then $\text{rad}(f) = p_1p_2\dots p_n$. The radical of a constant polynomial c is defined as $\text{rad}(c) = 1$.

2.2. Theorem Statement

Theorem 2.2.1 (The Mason-Stothers Theorem)**.** *Let a, b, and c be pairwise coprime polynomials over a field K with characteristic 0, such that $a + b + c = 0$. Assume that they are all non-zero and that they are not all constant. Then*

$$\max(\deg(a), \deg(b), \deg(c)) < \deg(\text{rad}(abc)).$$

I slightly modified the concluding statement of the Mason-Stohers theorem as compared to Snyder (2000) because of the implementation of degrees as natural numbers in Lean. There the conclusion was stated as $\max(\deg(a), \deg(b), \deg(c) \leq \deg(\text{rad}(abc))) - 1$. And if I would use the concluding statement from Snyder (2000), the condition that a, b, and c are not all constant would not be needed, because in the case that they are all constant we would get $0 \leq 0-1$, which holds for natural numbers in Lean, where we have $0 - 1 = 0$. In modifying the conclusion to $\max(\deg(a), \deg(b), \deg(c)) < \deg(\text{rad}(abc))$, we obtain a statement which would not hold if a, b, and c are constant. Furthermore, Snyder used $n_0(abc)$, which denotes the number of distinct zeros of the polynomial abc, instead of $\deg(\text{rad}(abc))$, but this is because he worked in the different setting of an algebraically closed field.

2.3. Overview of the Proof

In proving the Mason-Stothers theorem, I used the argumentation given by Snyder (2000), but with an adaptation inspired by Dahmen (2017). First, Snyder describes the Mason-Stothers theorem in the more restricted setting of an algebraically closed field (such as the complex numbers), while I work in the more general setting of a field with characteristic 0. Second, I factorize my proof further, compared with Snyder. At the highest level I factored the proof in 3 lemmas, a special case formulation of the Mason-Stothers theorem, and the Mason-Stothers theorem.

The first lemma, which is stated below, is the most complex part of the Mason-Stothers theorem. Its proof shows most of the mathematical tools that are needed.

Notations I denote $\gcd(a, b)$ by (a, b). The expression $\deg((a, b))$ is shortened to $\deg(a, b)$.

Lemma 2.3.1. *Let f be a polynomial over a field K, then*

$$\deg(f) \leq \deg(f, f') + \deg(\text{rad}(f)).$$

Proof. Write

$$f = u\, p_1^{a_1} p_2^{a_2} \dots p_n^{a_n},$$

with u a unit, the p_i's monic, irreducible and distinct, and the $a_i > 0 \in \mathbb{N}$. In the case f is zero the lemma trivially holds. Consider f is non-zero. Then $\forall p_i,\ p_i^{a_i-1} | f'$ (expand f' via the product rule) and $\forall p_i,\ p_i^{a_i-1} | f$, hence $p_i^{a_i-1} | (f, f')$. Because all p_i are distinct, monic, and irreducible, and we work over a UFD, we have

$$p_1^{a_1-1} p_2^{a_2-1} \dots p_n^{a_n-1} | (f, f').$$

Since $f \neq 0$ we have $(f, f') \neq 0$, and hence

$$\deg(p_1^{a_1-1} p_2^{a_2-1} \dots p_n^{a_n-1}) \leq \deg(f, f').$$

Also, we have that

$$\deg(f) = \deg(p_1^{a_1-1} p_2^{a_2-1} \dots p_n^{a_n-1}) + \deg(p_1 p_2 \dots p_n).$$

Combining these, we find that $\deg(f) \leq \deg(f, f') + \deg(\mathrm{rad}(f))$. □

In this proof we find the following mathematical machinery:

- polynomials;
- factorization of a polynomial in a unit and monic irreducible factors (polynomial normal form);
- properties of division by irreducible polynomials in a UFD;
- derivatives for polynomials;
- greatest common divisor of two polynomials;
- radical of a polynomial.

I now state the other lemmas and the theorems. The proofs of these are given in Section 4. They can mostly be proven when the aforementioned machinery is in place. In addition to the earlier mentioned mathematical machinery, we need the characteristic of a field in the proof of Mason-Stothers to be able to say that a polynomial is constant if it has zero derivative, which only holds for a field with characteristic 0.

Lemma 2.3.2. *Let f and g be coprime polynomials over a field K. And let the wronskian of f and g be zero, that is, $f'g - fg' = 0$. Then $f' = 0$ and $g' = 0$.*

Lemma 2.3.3. *Let f, g be polynomials over a field K. Then*

$$\deg(f'g - fg') \leq \deg(f) + \deg(g) - 1.$$

Theorem 2.3.4 (The Mason-Stothers Theorem: Special Case Formulation)**.** *Let $a, b,$ and c be pairwise coprime polynomials over a field K with characteristic 0, such that $a + b = c$. Assume that they are all non-zero and that they are not all constant. Then*

$$\deg(c) < \deg(\mathrm{rad}(abc)).$$

Theorem 2.2.1 (The Mason-Stothers Theorem)**.** *Let $a, b,$ and c be pairwise coprime polynomials over a field K with characteristic 0, such that $a + b + c = 0$. Assume that they are all non-zero and that they are not all constant. Then*

$$\max(\deg(a), \deg(b), \deg(c)) < \deg(\mathrm{rad}(abc)).$$

3. Mathematical Formalization and the Lean Proof Assistant

We can use computers to verify the correctness of proofs. The process of creating proofs that are so precise and complete that a computer can verify their correctness is called mathematical formalization. In practice one interacts with a proof assistant. The user of the proof assistant instructs the *proof engine*, which in turn creates a *proof object* (or formal proof) which is then checked by the *proof checker* that is part of the proof assistant (Geuvers, 2009). Figure 3.1 shows the interaction with a proof assistant.

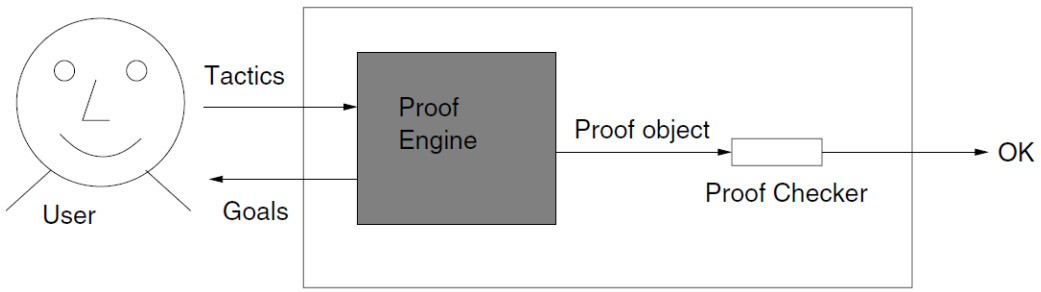

Figure 3.1.: The interaction of the user with a proof assistant. Reprinted from Geuvers (2009, p. 9).

3.1. The Process of Formalizing

The process of formalization (often) consists of three phases. In these phases we develop

1. a detailed document in LaTeX;
2. the definitions and statement of theorems in the proof assistant;
3. the proofs in the proof assistant.

The formalization starts with writing down the proof very detailed; this can be done in LaTeX. The level of detail in this document is much higher than an ordinary mathematical paper, because in an ordinary textbook proof some steps are still left to the reader. When this document is in place the development of the formal proof in the proof assistant starts. In the second phase we develop the definitions and statements of

the theorems. In the third phase we develop the proofs. This order gives a top-down approach to formalization, also known as the waterfall model. However, I took a more iterative approach, where the separation between phase 2 and 3 is less strict, and I repeatedly moved between phase 2 and 3. After finalizing the formalization, I created a new phase 1 document from scratch.

3.2. The Lean Proof Assistant

In this formalization project the Lean proof assistant was used. The motivation for using Lean was given in the introduction. In this section I describe Lean's logic and how to use Lean.

Logical foundation The logic that Lean uses is a form of typed lambda calculus (Church, 1932). In typed lambda calculus every term has a *type*. Via the Curry-Howard correspondence a logical statement can be encoded as a type (Howard, 1980). If a term in the typed lambda-calculus has as its type a logical statement, then this term represents a proof of the statement. Now proof checking becomes equivalent to type checking, a phenomenon known as the proofs-as-types paradigm.

Types In Lean every term has a type, as was already mentioned. For example `2 : ℕ`, expresses that the constant `2` is of type `ℕ`, which is the type of natural numbers. A function from ℕ to ℕ is expressed as `f : ℕ → ℕ`, here `f` has the type `ℕ → ℕ`. Even types have a type in Lean. The type of a type is called a *universe*. This creates a hierarchy of universes, where each universe is an element of a higher universe. Universes are denoted by `Type u`, where `u` is the universe level. The bottom universe is called `Prop`, and it is the universe were propositions live.

Lean allows for *dependent types*, also known as *Pi* types, where types can depend on terms. An example of a dependent type is `vector α n`, which is the type of vectors of elements of `α` of length `n`. The function `f : ℕ → ℕ` is an example where a type depends trivially on a term.

Lean supports *quotient types*. Where we start with a type `α` and a relation `r` on `α`, and we obtain the type `α` 'modulo' `r`.

3.3. Interacting with Lean

Lean allows for multiple approaches to instructing the proof engine. In Lean the *input language* is a combination of *procedural* and *declarative* statements. The procedural language elements are called *tactics*; these are commands that aim to advance in the *goal state*. Declarative statements resemble informal mathematical proofs more closely. Using tactics can be faster than using declarative statements, however declarative statements can give a clearer view on the structure of the proof. We often use a combination of both.

Interacting with the Lean theorem prover feels like functional programming. In particular the notation for function application differs from ordinary mathematics, as well as extensive use of the currying isomorphism (Avigad et al., 2018b). A mathematician would define a function like $f : X \times Y \to Z$, and states function application, for $x \in X$ and $y \in Y$, by $f(x, y)$. In Lean we express this as $f : X \to Y \to Z$ and the function application as $f\,x\,y$. Where the '$\to$' associates to the right, and function application associates to the left.

In writing mathematics the user can make the following declarations:

- *axioms*, which only have a type;
- *definitions*, which have a type and a value;
- *inductive families of types*, which is a mechanism for defining types.

Stating and proving a lemma or a theorem is equivalent to giving a definition, where the type of the definition is the statement of the lemma, and the proof of the lemma is the value of the definition. Using the inductive family of types we can construct types other than the universes, and the dependent types. In fact, every concrete type in Lean is either a type universe, dependent type, quotient type, inductive type or a combination of these.

To illustrate these different types of declarations I provide an example for each of the declarations above. An example axiom is the following:

```
axiom example_axiom : ∀x : ℕ, ∃ y : ℕ, y < x
```

This example illustrates that we must be careful in stating axioms, as they potentially lead to contradictions, as is the case with this example. An example definition is the following:

```
def f (x : ℕ) : ℕ := x + 7
```

Here we define the function `f`. It takes the single parameter `x` of type `ℕ`, and the type of the output is `ℕ`. The type of `f` is `ℕ -> ℕ`, and the value of `f` is `λ (x : ℕ), x + 7`. This last term is a *lambda abstraction*; these are used to build functions in lambda calculus. An example of an (simple) inductive family of types is the natural numbers:

```
inductive nat
| zero : nat
| succ (n : nat) : nat
```

Here the two constructors that can be used for creating an object of type `nat` are `zero` and `succ`.

4. Formalization

In this chapter I discuss the formalization of the Mason-Stothers theorem and the formalization of the background mathematics that was needed. The background mathematics forms a reusable library. I start with describing the background mathematics that was formalized.

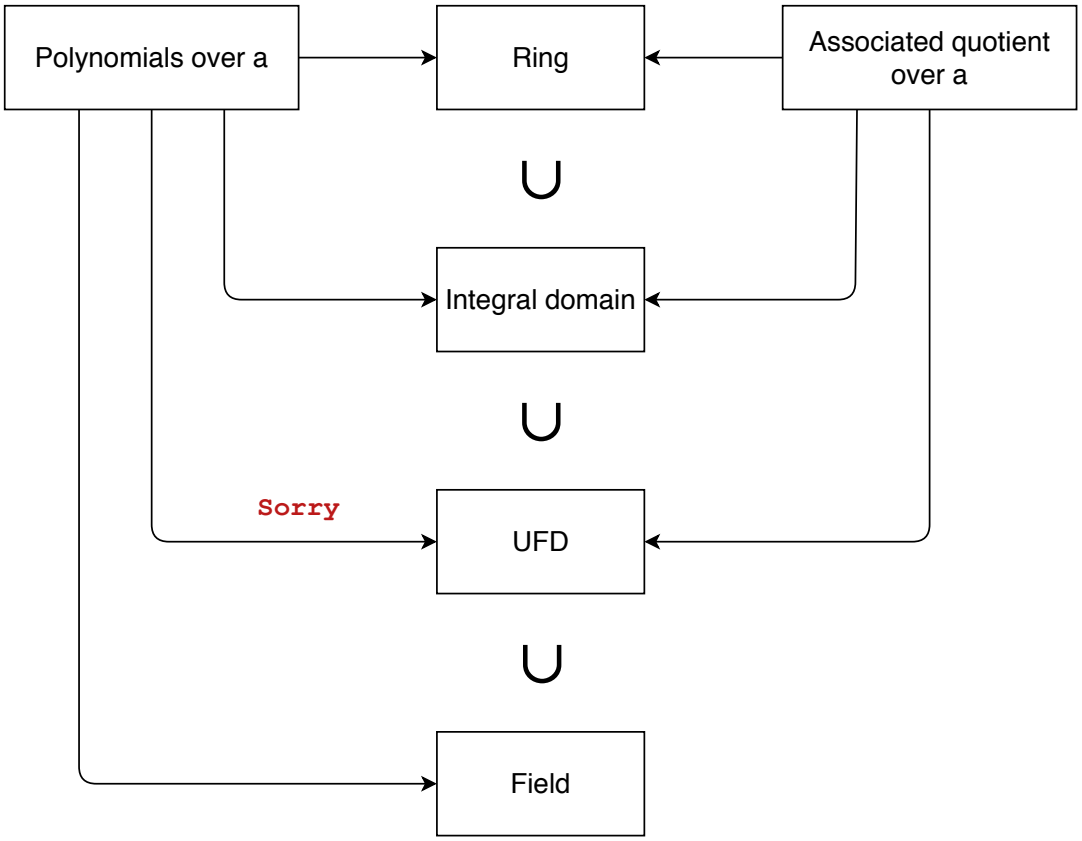

Figure 4.1.: Structure of the reusable library for ring theory.

Overview of the reusable library The reusable library developed contains mainly ring theory. Figure 4.1 shows an overview of the reusable library developed. Each box represents a component of the formalized library. In the middle we find the hierarchy of rings. On the sides we find the polynomials and the associated quotient that were developed for each type of ring. Lean's library for mathematics, Mathlib, already contained formalizations for rings, integral domains, and fields; I extended these formalizations.

A formalization of unique factorization domains, polynomials, and the associated quotient was built from scratch. The 'sorry' on the 'polynomials over a UFD' component indicates the single missing proof of the entire formalization of Mason-Stothers and the reusable library; it is described in Section 4.2.2.

Currently my formalization is situated in an independent repository (Wagemaker and Hölzl, 2018). The aim is that in time the formalization will be transferred to Mathlib.

4.1. Unique Factorization Domains

Here I describe my formalization of UFDs, using the definitions from (Dummit and Foote, 2004, p. 285). The Lean library already contained a type for rings, and for two special types of rings: fields and integral domains (the top and near-bottom of the ring hierarchy from Section 2.1).

To define UFDs we first classify certain ring elements.

Definition 4.1.1. Unit
We call $x \in R$ a *unit* if it has a multiplicative inverse.

Lean already had a type for units:

```
structure units (α : Type u) [monoid α] :=
  (val : α)
  (inv : α)
  (val_inv : val * inv = 1)
  (inv_val : inv * val = 1)
```

Here `structure` is used to create a non-recursive inductive type.

I introduced a predicate that denotes if an element of the ring is a unit, since this predicate is easier to work with. This `is_unit` predicate is defined as

```
def is_unit (a : α) : Prop := ∃b : units α, a = b
```

In this definition Lean performs a coercion on `b` that replaces `b` by `b.val`, which has type `α`, hence the equality `a = b` is well defined.

Definition 4.1.2. Irreducible
We call a non-zero non-unit element $y \in R$ *irreducible* if $y = ab$ implies a is a unit, or b is a unit, i.e. irreducible elements cannot be factored non-trivially.

I defined irreducible elements in Lean using:

```
def irreducible' [integral_domain α] (p : α) : Prop :=
  p ≠ 0 ∧
    ¬is_unit p ∧
      ∀a b : α, p = a * b → (is_unit a ∨ is_unit b)
```

Here the '`'` is used to discriminate with another equivalent definition of irreducibility.

The uniqueness in a UFD is up to association; we define association here.

Definition 4.1.3. Associated Relation
The elements $a, b \in R$ are *associated* if there exists a unit u such that $a = u * b$. I.e. a and b differ by a unit.

I defined it similarly in Lean:

```
def associated [integral_domain α] (x y : α) : Prop :=
  ∃u : units α, x = u * y
```

For the associated relation I introduced the infix notation `a ~ᵤ b`.

Definition 4.1.4. Unique Factorization Domain
A *Unique Factorization Domain* (UFD) is an integral domain with for each non-zero element r that is not a unit it holds that:

- r can be written as a *finite product of irreducible factors* p_i (not necessarily distinct): $r = p_1 p_2 \dots p_n$ and
- this decomposition is *unique up to associates and order*: namely if $r = q_1 q_2 ... q_m$ is another factorization of r into irreducibles, then $m = n$ and there is some renumbering of the factors so that p_i is associate to q_i for $i = 1, 2, \dots, n$.

I implemented the factorizations as finite multisets. The type `multiset` was already present in the library and represents finite multisets. The property that these factorizations are 'unique up to associates' uses that a relation (the association relation) holds for certain pairs, with one element in the multiset $\{p_1, p_2, \dots, p_n\}$, and the other element in the multiset $\{q_1, q_2, \dots, q_m\}$. I.e. the multisets $\{p_1, p_2, \dots, p_n\}$ and $\{q_1, q_2, \dots, q_m\}$ could be built by adding pairs (p_i, q_j), for which some relation $p_i R q_j$ holds, one by one. In other words, we are lifting a relation over elements to a relation over multisets. The construction of adding the pairs one by one leads to the following inductive definition of the lifting of a relation between multisets:

```
inductive rel_multiset {α β : Type u} (r : α → β → Prop) :
  multiset α → multiset β → Prop
| nil : rel_multiset ∅ ∅
| cons : ∀a b xs ys, r a b →
    rel_multiset xs ys →
      rel_multiset (a::xs) (b::ys)
```

Here the `cons` constructor adds the elements `a` and `b`, of the pair `(a,b)`, for which the relation `r` holds (`r a b`), to the multisets `xs` and `ys`. Adding an the element `a` to the multiset `xs` is denoted by `a::xs`.

Now we defined a UFD in Lean as:

```
class unique_factorization_domain (α : Type u)
extends integral_domain α :=
  (fac :
    ∀{x : α}, x ≠ 0 →
      ¬is_unit x →
        ∃p : multiset α, x = p.prod ∧ (∀x ∈ p, irreducible x))
  (unique :
    ∀{f g : multiset α},
      (∀x ∈ f, irreducible x) →
        (∀x ∈ g, irreducible x) →
          f.prod = g.prod →
            rel_multiset associated f g)
```

Here `fac` represents the property that every element can be factored (point 1 in Definition 4.1.4), and `unique` expresses the uniqueness up to associates of these factorizations (point 2 in the Definition 4.1.4).

4.1.1. Greatest Common Divisor

Definition 4.1.5. Divides Relation
Let R be a commutative ring, and let $a, b \in R$. We say that a *divides* b, notation $a|b$, if there exists a $c \in R$ such that $b = a * c$.

The divides relation was already implemented in Lean:

```
instance comm_semiring_has_dvd : has_dvd α :=
  has_dvd.mk (λa b, ∃c, b = a * c)
```

Definition 4.1.6. Greatest Common Divisor
Let R be a commutative ring, and let $a, b \in R$. Then we call $d \in R$ a *greatest common divisor* of a and b, notation $d = \gcd(a, b)$, if it satisfies the following axioms:

(i) $d|a$ (d divides a)

(ii) $d|b$

(iii) $\forall e \in R$ with $e|a$ and $e|b$, we have $e|d$

In general the gcd is not unique. It is, however, unique up to associates. Note that over $\mathbb{Z}$ we can have the additional axiom that the gcd is non-negative, this would lead to a unique gcd for any two elements. However, we cannot do this in an arbitrary ring.

I took a computational approach in defining the gcd. I defined it as a function that produces an element that has the desired properties. Given a ring R, we defined gcd as a function $\gcd : R^2 \to R$ (with the aforementioned properties).

```
class has_gcd (α : Type u) [comm_semiring α] :=
  (gcd : α → α → α)
  (gcd_right : ∀a b, gcd a b ∣ b)
  (gcd_left  : ∀a b, gcd a b ∣ a)
  (gcd_min   : ∀a b g, g ∣ a → g ∣ b → g ∣ gcd a b)
```

Definition 4.1.7. Coprime
Let R be a ring, which has a gcd for each pair of elements, then we call $a, b \in R$ *coprime* if $\gcd(a, b)$ is a unit.

This translated directly in Lean.

```
def coprime (a b : α) := is_unit (gcd a b)
```

In UFDs every pair of elements has a gcd. If we let a and b be two non-zero elements with factorizations $a = u\,p_1^{e_1}p_2^{e_2}\dots p_n^{e_n}$ and $b = v\,p_1^{f_1}p_2^{f_2}\dots p_n^{f_n}$, where the p_i are irreducible, distinct, and not-associated, and u and v are units, then

$$d = p_1^{\min(e_1,f_1)}p_2^{\min(e_2,f_2)}\dots p_n^{\min(e_n,f_n)}$$

is a gcd of a and b.

However obtaining these specific representations of a and b, where the irreducible factors are distinct and not-associated is non-trivial. Suppose we wanted to obtain them, because we are in a UFD we can first get the factorizations $a = p_1p_2\dots p_n$ and $b = q_1q_2\dots q_m$, and then we would have to define an algorithm that transfers these factorizations to factorizations with distinct and not-associated factors. Formalizing this algorithm would be lengthy, and then we would still have to prove that the d given above is actually a gcd of a and b.

Instead of using an algorithm to obtain the non-associated factors, I took a more direct approach were I could avoid these representations. In this direct approach I used the intersection operation on (finite) multisets. The intersection on multisets can be explained as follows: we see multisets as functions from the ring to $\mathbb{N}$, then for any two multisets f, and g, we have $(f \cap g)(x) = \min(f(x), g(x))$. The idea is to apply this intersection operation to the multisets with the factors in a and b, to obtain the gcd. That is if $a = p_1p_2\dots p_n$ and $b = q_1q_2\dots q_m$, that we take the intersection of the multisets: $\{p_1, p_2, \dots, p_n\} \cap \{q_1, q_2, \dots, q_m\}$. However this would not work if we had factors p_i which would be distinct but associated. E.g. in $\mathbb{Z}$, we have $2 \sim_u -2$, and we could factorize $4 = 2 * 2$ or $4 = (-2) * (-2)$, we know that $\gcd(4, 4) = 4$, but if we intersect the factorizations, we would get $\{2, 2\} \cap \{-2, -2\} = \emptyset$.

To circumvent this problem with distinct but associated irreducible factors, I developed the *associated quotient* structure, which is the quotient obtained of the ring with respect to the associated relation (which is an equivalence relation). This structure has the nice property that two elements are equal if and only if they are associated, hence the case of having distinct but associated elements can never occur here.

4.1.2. The Associated Quotient

The associated quotient is obtained by considering the elements of a ring modulo association. Recall that we defined association ($a \sim_u b$) for elements $a, b \in \alpha$ by $\exists u : \text{units } \alpha,\ a = u * b$, i.e. a and b differ by a unit.

Definition 4.1.8. Associated Quotient
Let R be a commutative ring. Then the *associated quotient* of R, denoted by $R/\sim_u$, is the set of all equivalence classes with respect to the associated relation.

In Lean I defined the type `quot` to denote the associated quotient. To obtain this quotient type in Lean we first form a *setoid*, which is a set (the carrier set of the commutative ring) equipped with an equivalence relation (association).

```
def setoid (β : Type u) [comm_semiring β] : setoid β :=
  {r := associated, iseqv := associated.eqv}
```

From this setoid we create a quotient type:

```
def quot (β : Type u) [comm_semiring β] : Type u :=
  quotient (setoid β)
```

Now `quot` is the type of the associated quotient.

As an algebraic structure the associated quotient is a commutative monoid with 1 and 0. The binary operation is the multiplication that is lifted from the ring to the associated quotient. The divisibility relation ($a \mid b$) gives the associated relation a *preorder*.

Associated quotient over integral domains On integrals domains the preorder is extended to a *partial order*, as we can prove the anti-symmetry ($a \leq b \wedge b \leq a \Rightarrow a = b$).

Associated quotient over UFDs This partial ordering could be extended to a bounded lattice, when working over UFDs. The bottom of this lattice is $\bot = 1$, because 1 divides every element. The top of this lattice is $\top = 0$, because every element divides 0. The supremum is the greatest common divisor and the infimum is the least common multiple (lcm).

To prove the existence of a gcd in UFDs, I first proved its existence in the associated quotient over a UFD. In order to do this, I lifted the notion of irreducibility and the factorization of the UFD to the quotient. The lifting of the irreducible predicate is called `irred`. The lifting of the property that every element can be factorized in irreducible elements (point 2 in Definition 4.1.4) is done via the lemma `representation` below.

```
lemma representation (a' : quot α) : a' ≠ 0 →
  ∃p : multiset (quot α), (∀a ∈ p, irred a) ∧ a' = p.prod
```

From this statement asserting the existence of a factorization in irreducible factors, we can obtain an actual multiset, with the factors dividing the element, by using the axiom of choice. This is done in Lean using `some`. I defined the function that takes an element and gives a multiset of irreducible factors, named `to_multiset`, below.

```
def to_multiset (a : quot α) : multiset (quot α) :=
  if h : a = 0 then 0 else classical.some (representation a h)
```

With this `to_multiset` function I defined a sup and inf (or meet and join) to obtain a lattice structure for the associated quotient.

```
def inf (a b : quot α) :=
  if a = 0 then b
  else if b = 0 then a
  else (to_multiset a ∩ to_multiset b).prod
def sup (a b : quot α) :=
  if a = 0 then 0
  else if b = 0 then 0
  else (to_multiset a ∪ to_multiset b).prod
```

Here the ∩ and ∪ operators are working on multisets, as described earlier. Once these definitions for `sup` and `inf` were in place, the proof that they had the desired properties (of being gcd and lcm) was not difficult.

Now to obtain the gcd of two elements a, b in the UFD, we take the inf of the equivalence classes of a, and b, $(inf([a], [b]))$, and take a representative for this equivalence class.

```
gcd := assume a b : α, quot.out (inf (mk a) (mk b))
```

Here `α` is the type of the UFD, and `quot.out` takes the representative, and with `mk a`, we obtain the equivalence class of `a`.

4.2. Polynomials

The polynomial library I develop is a continuation of the work from Hölzl (2017), which is an initial write-up, containing a definition for polynomials, an induction rule, and basic lemmas for degrees and derivatives.

Definition 4.2.1. Polynomial
We define $R[X]$, the *polynomials* over R, to be the set of all function from $\mathbb{N}$ to R with finite support.

We can see polynomials as infinite lists with only finitely many non-zero terms, i.e. $f = (a_0, a_1, \dots, a_n, 0, 0, \dots)$ is a polynomial, such that for all $m > n$ we have that $a_m = 0$. Here $f(n) = a_n$ for all $n \in \mathbb{N}$, and hence f is a function from $\mathbb{N}$ to $\mathbb{R}$.

Lean already contained a library for describing functions with a finite support.

```
def finsupp (α : Type u) (β : Type v) [has_zero β] :=
  {f : α → β // finite {a | f a ≠ 0}}
```

It is implemented as a *subtype*. In a subtype we take a type and add an extra condition or property for the elements of this type; it is an example of a non-recursive inductive type. We can create an element of a subtype by providing an element, and the proof that this element has the property needed for the subtype. The definition of `finsupp` can be understood as the type of functions from α to β, (`f : α → β`), for which the set of elements in the domain that are mapped to a non-zero value is finite, (`finite {a | f a ≠ 0}`). For finite support functions from α to β the infix notation `α →₀ β` is used.

The polynomial definition is now:

```
def polynomial (α : Type u) [semiring α] := ℕ →₀ α
```

We can represent polynomials using the following formal expression: $f = a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$, as was discussed in Section 2.1, which is convenient for making definitions using polynomials.

The smallest polynomials are the *monomials*, which are only non-zero for a single element in their domain.

Definition 4.2.2. Monomial
We define *monomials* as polynomials of the form aX^n, for some $a \in R$ and $n \in \mathbb{N}$. Monomials of the form aX^0, or simply a, are called *constant* polynomials.

For the `finsupp` type (and hence for the polynomials) these 'smallest' functions are called `singles`, and they are defined as:

```
def single (a : α) (b : β) : α →₀ β :=
  ⟨λa', if a = a' then b else 0,  proof⟩
```

Here `a` is an element in the domain, and `b` is an element in the codomain.

Two special monomials are X, and Ca the constant polynomial with value a. These are defined in Lean as:

```
def X : polynomial α := finsupp.single 1 1
def C (a : α) : polynomial α := finsupp.single 0 a
```

I defined a predicate that indicates if a polynomial is a constant polynomial.

```
def is_constant (p : polynomial α) : Prop := ∃ c : α, p = C c
```

To work with polynomials, and to define functions on polynomials (and finite support functions), we need to be able to destruct them, and extract the information they contain. This is done using the sum operator that is available for finite support functions. Given a function $f : \alpha \to \beta$ with finite support, and a function $g : \alpha \times \beta \to \gamma$. Then we can compute $\Sigma_{a \in \text{support}(f)}\ g(a, f(a))$ using the sum operator. This sum operator is defined in Lean as:

```
def sum [has_zero β] [add_comm_monoid γ] (f : α →₀ β)
  (g : α → β → γ) : γ :=
    f.support.sum (λa, g a (f a))
```

To illustrate the sum operator I use it to represent polynomials in a form similar to $f = a_0 + a_1X^1 + a_2X^2 + \cdots + a_nX^n$. For this we use the `sum` operator and the special polynomials `X` and `C`.

```
lemma sum_const_mul_pow_X  {f : polynomial α} :
  finsupp.sum f (λn a, C a * X ^ n) = f
```

Now I give definitions for the two binary operations that give polynomials (and finite support functions) their ring structure (Dummit and Foote, 2004, p. 295). These definitions were already implemented in Lean.

Definition 4.2.3. Polynomial Addition
The *addition of polynomials* is 'componentwise':

$$\sum_{i=0}^{n} a_iX^i + \sum_{i=0}^{n} b_iX^i = \sum_{i=0}^{n} (a_i + b_i)X^i$$

(here a_n or b_n may be zero in order for addition of polynomials of different degrees to be defined).

The addition was implemented in Lean by *zipping* the addition of R.

```
instance [add_monoid β] : has_add (α →₀ β) :=
  ⟨zip_with (+) (add_zero 0)⟩
```

Definition 4.2.4. Polynomial Multiplication
The *multiplication of polynomials* is performed by first defining $(aX^i)(bX^j) = abX^{i+j}$ and then extending to all polynomials by the distributive laws, so that in general we have

$$(\sum_{i=0}^{n} a_iX^i) \times (\sum_{i=0}^{n} b_iX^i) = \sum_{k=0}^{n+m} (\sum_{i=0}^{k} a_ib_{k-i})X^k.$$

The multiplication was implemented using the `sum` operator. Let f and g be functions with finite support, then we can write the above multiplication definition in terms of the `sum` operator as

$$\begin{aligned} f \times g :=& \Sigma_{\substack{a \in \text{support}(f) \\ b \in \text{support}(g)}} (f(a) \times g(b))X^{a+b} \\ =& \Sigma_{\substack{a \in \text{support}(f) \\ b \in \text{support}(g)}} \text{single}(a+b)(f(a) \times g(b)) \end{aligned}$$

In Lean this gives

```
instance [has_add α] [semiring β] : has_mul (α →₀ β) :=
  ⟨λf g, f.sum $ λa₁ b₁, g.sum $ λa₂ b₂,
    single (a₁ + a₂) (b₁ * b₂)⟩
```

Here the $ sign is used to reduce the amount of parentheses; It means that everything on the right of $ should be grouped.

For a polynomial we identify the following features.

Definition 4.2.5. Polynomial Features
Let $f = a_0 + a_1X + a_2X^2 + \cdots + a_nX^n \in R[X]$. If $a_n \neq 0$ then f is of *degree* n, a_nX^n is the *leading term*, and a_n is the *leading coefficient*. The polynomial is called *monic* if $a_n = 1$.

To make these definitions in the formalization we must work a bit harder. We first define the supremum of a finite set of elements of type α, where α is equipped with a lattice structure, by folding the join (⊔) of the lattice over the finite set. The bottom of the lattice (⊥) is used to start the fold.

```
variables {α : Type u} {β : Type w}
  [decidable_eq β] [semilattice_sup_bot α]
def Sup_fin (s : finset β) (f : β → α) : α := s.fold (⊔) ⊥ f
```

Now the degree was defined as the supremum of the support of the polynomial.

```
def degree (p : polynomial α) : ℕ := p.support.Sup_fin id
```

With the degree defined, I defined the leading coefficient as the polynomial evaluated at its degree, where this evaluation is with regard to the mapping $\mathbb{N} \to R$.

```
def leading_coeff (p : polynomial α) : α := p (degree p)
```

The property of being monic translated directly.

```
def monic (p : polynomial α):= leading_coeff p = 1
```

4.2.1. Polynomials over Integral Domains

An integral domain is a commutative ring with $0 \neq 1$ and no zero-divisors, as was described in Section 2.1. It is well known that a polynomial ring over an integral domain is an integral domain. I proved this using the leading coefficients of a polynomial.

```
lemma zero_ne_one : (0 : polynomial α) ≠ 1

lemma eq_zero_or_eq_zero_of_mul_eq_zero :
  ∀f g : polynomial α, f * g = 0 → f = 0 ∨ g = 0
```

Formal derivative of polynomials

Definition 4.2.6. Formal Derivative of Polynomial
Let $f = a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$ be a polynomial, then the *derivative* of f is $f' = a_1 + 2a_2X + 3a_3X^2 + \cdots + a_nnX^{n-1}$.

The derivative was implemented using the `sum` operator:

$$f' = \Sigma_{n \in \text{support}(f)}(\begin{cases} 0 & \text{if } n = 0 \\ (m+1)f(m+1)X^m & \text{if } n = m+1 \end{cases})$$

Which translates in Lean to:

```
def derivative (p : polynomial α) : polynomial α :=
  p.sum (λn a, nat.cases_on n 0 (λn, single n (a * (n + 1))))
```

Here `nat.cases_on` is used to make a case distinction on the natural number n. For the derivative of f in Lean I introduce the notation `d[f]`.

Most notably the *product rule* for differentiation is used in the proof of the Mason-Stothers theorem.

```
lemma derivative_mul {f : polynomial α} :
  ∀{g}, derivative (f * g) =
    derivative f * g + f * derivative g
```

4.2.2. Polynomials over Unique Factorization Domains

It can be proven that a polynomial ring over a UFD is itself a UFD. The proof of this is statement is the only missing proof in the entire formalization of Mason-Stothers, the proof of Mason-Stothers, and the background mathematics that was formalized; it has to be proved in later work to finalize the formalization of the Mason-Stothers theorem. A missing proof in Lean is indicated by **`sorry`**. It has the effect of adding an axiom.

```
instance {α : Type u} [unique_factorization_domain α] :
  unique_factorization_domain (polynomial α) :=
    {fac := sorry,
     unique := sorry,
     .. polynomial.integral_domain}
```

4.2.3. Polynomials over Fields

Monic polynomials In a polynomial ring over a field, each non-zero polynomial can be made monic by multiplying with the inverse of the leading coefficient.

```
def make_monic [field α] (f : polynomial α) :=
  if (f = 0) then 0 else (C (f.leading_coeff)⁻¹ * f)
```

By lifting `make_monic` to the associated quotient, we can always choose the unique monic representative.

```
lemma monic_monic_out_of_ne_zero [field α]
  (f : quot (polynomial α)) (h : f ≠ 0) :
    monic (monic_out f)
```

Here `monic_out` is obtained by lifting `make_monic`.

Polynomial normal form In a field we can write f in a normal form. I have proven that a field is a UFD (which is trivial, since each element not equal to zero is a unit). And as we took as an axiom that a polynomial ring over a UFD is itself a UFD, we have that a polynomial ring over a field is a UFD. Hence we can obtain a factorization of f as a product of irreducible factors. Now by grouping all the leading coefficients of the irreducible factors of f we obtain the normal form.

Definition 4.2.7. Polynomial Normal Form
Let K be a field, and $f \in K[X]$ a polynomial. Then the *normal form* of f is given by $f = c\ p_1 p_2 \dots p_n$, where c is a constant polynomial, and the p_i's are all monic and irreducible. In the normal form we decompose f into a constant factor and a multiset of monic irreducible factors. In the case that $f = 0$, we set the multiset of monic irreducible factors equal to the empty set, which then gives a unique normal form.

I proved the existence of this normal form.

```
lemma polynomial_fac [field α] (x : polynomial α) :
  ∃c :  α, ∃p : multiset (polynomial α), x = C c * p.prod ∧
    (∀x ∈ p, irreducible x ∧ monic x)
```

To practically work with this normal form of f, I used choice to obtain the factors and the constant from the above existential lemma.

```
def c_fac (p : polynomial α) : α :=
  if (p = 0) then 0 else some (polynomial_fac p)

def factors (p : polynomial α) : multiset (polynomial α) :=
  if (p = 0)
    then 0 else classical.some (some_spec $ polynomial_fac p)
```

The term `factors` comes with the accompanying lemmas that state these factors are monic and irreducible.

```
lemma factors_monic (p : polynomial α) :
  ∀x ∈ p.factors, monic x

lemma factors_irred (p : polynomial α) :
  ∀x ∈ p.factors, irreducible x
```

To finalize the normal form we write f using the constant factor and the monic irreducible factors.

```
lemma factors_eq (p : polynomial α) :
  p = C p.c_fac * p.factors.prod
```

Radical of a polynomial

Definition 4.2.8. Radical
Let K be a field and f be a polynomial over K. Write f in an adapted normal form, were we group equal factors of p_i into factors of the form $p_i^{a_i}$, $f = c\,p_1^{a_1}p_2^{a_2}\dots p_n^{a_n}$, with c a constant polynomial, and the p_i's monic and irreducible, the $a_i > 0$ for all i, and in addition we assume that the p_i's are distinct, i.e. $p_i \neq p_j$ if $i \neq j$. Then the *radical* of f is $\mathrm{rad}(f) = p_1p_2\dots p_n$. In other words, we take the product of the distinct monic irreducible factors dividing f. The radical of a constant polynomial c is defined as $\mathrm{rad}(c) = 1$.

In Lean I defined the radical by removing the duplicate factors in `factors`.

```
def rad (p : polynomial β) : polynomial β :=
  p.factors.erase_dup.prod
```

Field of characteristic zero

Definition 4.2.9. Characteristic zero
Let K be a field, and let $e : \mathbb{Z} \to K$ be the unique map from $\mathbb{Z}$ to K that is compatible with the addition and multiplication. If for all $n \in \mathbb{Z}$ with $n \neq 0$ it holds that $e(n) \neq 0$, then K is said to have *characteristic zero*.

For implementing the property of characteristic zero, it is sufficient to only consider $n \in \mathbb{N}$ due to properties of the map e (it is a ring homomorphism).

```
def characteristic_zero (α : Type u) [semiring α] : Prop :=
  ∀n : ℕ, n ≠ 0 → (↑n : α) ≠ 0
```

Here the '↑' is the map e function; it acts as a coercion from `ℕ` to `α`.

4.3. The Proof of the Mason-Stothers Theorem

In this section I give the informal proof of the Mason-Stothers theorem; the formalization follows this proof. I state the formal translations of the lemmas, and I highlight some of the steps in the formal proof.

For completeness, I again state the proof of the first lemma.

Lemma 2.3.1. *Let f be a polynomial over a field K, then*

$$\deg(f) \leq \deg(f, f') + \deg(\mathrm{rad}(f)).$$

Proof. Write

$$f = u\,p_1^{a_1}p_2^{a_2}\dots p_n^{a_n},$$

with u a unit, the p_i's monic, irreducible and distinct, and the $a_i > 0 \in \mathbb{N}$. In the case f is zero the lemma trivially holds. Consider f is non-zero. Then $\forall p_i$, $p_i^{a_i-1} | f'$ (expand

f' via the product rule) and $\forall p_i,\ p_i^{a_i-1}|f$, hence $p_i^{a_i-1}|(f,f')$. Because all p_i are distinct, monic, and irreducible, and we work over a UFD, we have

$$p_1^{a_1-1}p_2^{a_2-1}\dots p_n^{a_n-1}|(f,f').$$

Since $f \neq 0$ we have $(f,f') \neq 0$, and hence

$$\deg(p_1^{a_1-1}p_2^{a_2-1}\dots p_n^{a_n-1}) \leq \deg(f,f').$$

Also, we have that

$$\deg(f) = \deg(p_1^{a_1-1}p_2^{a_2-1}\dots p_n^{a_n-1}) + \deg(p_1p_2\dots p_n).$$

Combining these, we find that $\deg(f) \leq \deg(f,f') + \deg(\mathrm{rad}(f))$. □

The statement of Lemma 2.3.1 translated in Lean as:

```
lemma Mason_Stothers_lemma (f : polynomial β) :
  degree f ≤
    degree (gcd f (derivative f )) + degree (rad f) := proof
```

In the first step of the proof we write f in the polynomial normal form that I defined in Section 4.2.3.

The intermediate step $\forall p_i,\ p_i^{a_i-1}|f'$ was stated as a the following lemma:

```
private lemma Mason_Stothers_lemma_aux_1 (f : polynomial β) :
  ∀x ∈ f.factors,
    x^(count x f.factors - 1) ∣ d[f.factors.prod] := proof
```

We had to apply the product rule for derivatives to f'. This required a generalization to the earlier described product rule for derivatives:

```
lemma derivative_prod_multiset
  {s : multiset (polynomial α)} :
    derivative (s.prod) =
      (s.map (λb, derivative (b) * (s.erase b).prod)).sum :=
        proof
```

The division properties of irreducible elements in a UFD come into play at the step where we go from: $\forall i, p_i^{a_i-1}|(f,f')$ to $p_1^{a_1-1}p_2^{a_2-1}\dots p_n^{a_n-1}|(f,f')$. This statement followed from a general statements in UFDs. Below α is a type with a UFD structure.

```
lemma facs_to_pow_prod_dvd_multiset {s : multiset α} {z : α}
  (h1 : ∀x ∈ s, irreducible x ∧ (x^(count x s)) ∣ z ∧
      ∀y ∈ s, x ≠ y → ¬ (x ~ᵤ y)) :
        s.prod ∣ z := proof
```

This division property, of irreducible elements in a UFD, is also found in division by prime numbers in $\mathbb{N}$. In $\mathbb{N}$ we have that $2|a$ and $5|a$ implies $2 * 5|a$, because 2 and 5 are prime. There we also have $2^n|a$ and $5^m|a$ implies $2^n * 5^m|a$. In a general UFD we have to take into account association. E.g. over $\mathbb{Z}$ we don't have $2|a$ and $-2|a$ implies $-2 * 2|a$, because 2 and -2 are associated. Therefore we have the clause `∀y ∈ s, x ≠ y → ¬ (x ~ᵤ y)`.

Lemma 2.3.2. *Let f and g be coprime polynomials over a field K. And let the wronskian of f and g be zero, that is, $f'g - fg' = 0$. Then $f' = 0$ and $g' = 0$.*

Proof. Rewriting to $f'g = fg'$, we find that $f|f'g$, and $g|fg'$. Now because f and g are coprime, and the polynomial ring over K is a UFD, we have that $f|f'$ and $g|g'$. This only holds if both $f' = 0$, and $g' = 0$. □

Lemma 2.3.2 translated in Lean as:

```
lemma derivative_eq_zero_and_derivative_-
eq_zero_of_coprime_of_wron_eq_zero
  {a b : polynomial β}
  (h1 : coprime a b)
  (h2 : d[a] * b - a * d[b] = 0) :
    d[a] = 0 ∧  d[b] = 0 := proof
```

In the proof we again make use of the special division properties in a UFD:

```
lemma dvd_of_dvd_mul_of_coprime {a b c : α} (h1 : a ∣ b * c)
  (h2 : coprime a b) :
    a ∣ c := proof
```

Lemma 2.3.3. *Let f, g be polynomials over a field K. Then*

$$\deg(f'g - fg') \leq \deg(f) + \deg(g) - 1.$$

Proof. We make four case distinctions:

1	$\deg(f) = 0$	$g' = 0$
2	”	$g' \neq 0$
3	$\deg(f) \neq 0$	$\deg(g) = 0$
4	”	$\deg(g) \neq 0$

In each case we simplify using the the properties of degree. □

Remark. *The case where f and g are constant holds because the degree was defined as a natural number (where $0 - 1 = 0$).*

Lemma 2.3.3 translated in Lean as:

```
lemma degree_wron_le {a b : polynomial β} :
  degree (d[a] * b - a * d[b]) ≤ degree a + degree b - 1 :=
    proof
```

Theorem 2.3.4 (The Mason-Stothers Theorem: Special Case Formulation)**.** *Let $a, b,$ and c be pairwise coprime polynomials over a field K with characteristic 0, such that $a + b = c$. Assume that they are all non-zero and that they are not all constant. Then*

$$\deg(c) < \deg(\mathrm{rad}(abc)).$$

Remark. *In this special case of the formulation we set $a+b = c$, whereas in the general case we have $a + b + c = 0$.*

Proof. Via the sum rule for derivatives we have that $a' + b' = c'$. Multiplying the first equation by a', the second by a, and subtracting, we find

$$a'b - ab' = a'c - ac'. \tag{4.1}$$

Since $(a, a')|a$ and $(a, a')|a'$, we have $(a, a')|a'b - ab'$. Similarly $(b, b')|a'b - ab'$, and by equation (4.1) also $(c, c')|a'b - ab'$. Since a, b and c are coprime, we have that (a, a'), (b, b') and (c, c') are also coprime. And because we are in a UFD, we obtain

$$(a, a')(b, b')(c, c')|a'b - ab'. \tag{4.2}$$

We prove by contradiction that $a'b - ab'$ is non-zero. Assume that $a'b - ab' = 0$, then by equation (4.1) also $a'c - ac' = 0$, now we apply the Lemma 2.3.2 twice to get $a' = b' = c' = 0$, which is a contradiction since we have that not all a, b, and c are constant. And in a field of characteristic 0, a polynomial is constant if and only if its derivative is zero.

Since we now have that $a'b - ab'$ is non-zero, we can use equation (4.2) to obtain

$$\deg((a, a')(b, b')(c, c')) \leq \deg(a'b - ab'). \tag{4.3}$$

Also because $a'b-ab'$ is non-zero we have that $(a, a')(b, b')(c, c')$ is non-zero. Now because we work in an integral domain we have

$$\deg((a, a')(b, b')(c, c')) = \deg(a, a') + \deg(b, b') + \deg(c, c'). \tag{4.4}$$

By combining equations (4.3) and (4.4) with the Lemma 2.3.3 we obtain

$$\deg(a, a') + \deg(b, b') + \deg(c, c') \leq \deg(a) + \deg(b) - 1.$$

By adding $\deg(c)$ on both sides and rewriting, we obtain

$$\deg(c) \leq \deg(a) - \deg(a, a') + \deg(b) - \deg(b, b') + \deg(c) - \deg(c, c') - 1.$$

Now we apply Lemma 2.3.1 three times to get:

$$\begin{aligned} \deg(c) &\leq \deg(\mathrm{rad}(a)) + \deg(\mathrm{rad}(b)) + \deg(\mathrm{rad}(c)) - 1 \\ &\leq \deg(\mathrm{rad}(abc)) - 1. \end{aligned}$$

Taking the 1 to the other side allows us to conclude that

$$\deg(c) < \deg(\mathrm{rad}(abc)).$$

□

The Lean translation of this special case formulation is left out because it is similar to the general case formulation that follows. The Lean code of the proof just presented, is placed in appendix B. Several auxiliary lemmas that are used in that proof are not included in the appendix.

Theorem 2.2.1 (The Mason-Stothers Theorem)**.** *Let a, b, and c be pairwise coprime polynomials over a field K with characteristic 0, such that $a + b + c = 0$. Assume that they are all non-zero and that they are not all constant. Then*

$$\max(\deg(a), \deg(b), \deg(c)) < \deg(\mathrm{rad}(abc)).$$

Proof. We rewrite $a+b+c=0$ to $a+b=-c$, and then we use Theorem 2.3.4 to obtain $\deg(-c) < \deg(\mathrm{rad}(ab(-c)))$. Directly it follows $\deg(c) < \deg(\mathrm{rad}(abc))$. Similarly we obtain these expressions for a and b. The conclusion follows by definition of the maximum. □

I formulated the Mason-Stothers theorem in Lean as

```
theorem Mason_Stothers [field β]
  (h_char : characteristic_zero β)
  (a b c : polynomial β)
  (ha : a ≠ 0)
  (hb : b ≠ 0)
  (hc : c ≠ 0)
  (h_coprime_ab : coprime a b)
  (h_coprime_bc : coprime b c)
  (h_coprime_ca : coprime c a)
  (h_add : a + b + c = 0)
  (h_constant :
    ¬(is_constant a ∧ is_constant b ∧ is_constant c)) :
  max (degree a) (max (degree b) (degree c))
    < degree (rad (a*b*c))
```

5. Comparison to the Isabelle Formalization

Independent formalizations of the Mason-Stothers theorem have been performed recently in the Isabelle proof assistant (Eberl, 2017) and the Coq proof assistant (Mahboubi, 2018). In this chapter I compare my formalization in Lean to the formalization in Isabelle.

5.1. Unique Factorization Domains

In Isabelle a UFD is defined (Thiemann and Yamada, 2016) using a *factorial-monoid* (Ballarin et al., 2017). A *monoid* is an algebraic structure similar to a group, but without the inverses. An example of a monoid is a ring restricted to the multiplication. Now a factorial monoid has the same factorization properties as a UFD. Having a factorial-monoid gives a nice fine-structuring in the algebraic hierarchy. In Isabelle a UFD is defined as an integral domain R where the $R\backslash\{0\}$ forms a factorial-monoid.

In proving the existence of a gcd in a UFD, Thiemann and Yamada (2016) also made use of the associated quotient. But in comparison I make a more direct use, as I first prove that gcds exist in the associated quotient. The Isabelle formalization makes less direct use of the associated quotient, which is possibly because quotients are cumbersome to work with in Isabelle. Their idea for obtaining the gcd is similar to my idea, as they also take the intersection of the equivalence classes of the irreducible factors:

"∃c cs. c ∈carrier G ∧set cs ⊆carrier G ∧wfactors G cs c ∧
fmset G cs = fmset G as #∩ fmset G bs"

Here they want to obtain $\gcd(a, b)$. In this statement *as* and *bs* are the multisets with factors obtained from the factorization assumption, with *fmset* the factors in these multisets are mapped to their equivalence classes, *wfactors G cs c* states that the product of all the factors in *cs* is associated to *c*. Here the term *carrier* refers to the carrier set of the monoid. In Lean we do not use explicit carrier sets, because there the carrier set is the set of all elements of a type; this is possible in Lean as types are more flexible.

5.2. Polynomials

The polynomial formalization that was used in the Mason-Stothers theorem was given by Huffman et al. (2016). They define polynomials as function from $\mathbb{N}$ to some type

which has a zero, together with the statement that for all except finitely many n we have $f(n) = 0$.

typedef *'a poly = "{f :: nat ⇒ 'a::zero. ∀∞n. f n = 0}"*

This type is similar to our polynomial type definition; the difference is that we were more general by first defining functions with finite support, and then taking polynomials as a special case.

To practically work with polynomials, they defined a list style constructor and scalar multiplication for polynomials:

lift_definition *pCons :: "'a::zero ⇒ 'a poly ⇒ 'a poly"*
is *"λa p. case_nat a (coeff p)"*

If p is a polynomial, then $\text{pCons}(a, p) = pX + a$, so it shifts the polynomial.

lift_definition *smult :: "'a::comm_semiring_0 ⇒ 'a poly ⇒ 'a poly"*
is *"λa p n. a * coeff p n"*

Based on these two constructs, they can define polynomial multiplication:

definition *"p * q =*
fold_coeffs (λa p. smult a q + pCons 0 p) p 0"

The main difference, in working with polynomials, is that we put the emphasis on the support, whereas support is not even mentioned in the Isabelle formalization. They see polynomials as lists, which can be manipulated by the *smult* and *pcons* operator, and we see them as sums (over the support) of monomials, which can be manipulated using the `sum` operator. The viewpoint of sums of monomials (or singles in the case of an arbitrary finite support function) is more general because it can be applied to any function with a finite support, while the viewpoint of a list is not suited for arbitrary functions with finite support (they could be seen as a finite set of tuples).

In working with support and monomials (in particularly the `C a` and `X`) we not only provide a more general view on polynomials, but also a more intuitive view. That our representation of polynomials is intuitive to work with is showcased by our induction rule:

```
lemma induction_on
  {M : polynomial α → Prop} (p : polynomial α)
  (M_C : ∀(a : α), M (C a))
  (M_add : ∀{p q}, M p → M q → M (p + q))
  (M_mul_X : ∀{p}, M p → M (p * X) ) :
     M p
```

5.3. The Mason-Stothers Theorem

The Isabelle formalization of the Mason-Stothers theorem, contains an interesting factorization of the proof. The proof is based on the proof stated in Lemmermeyer (2005),

which in turn, is based on Snyder (2000). They first proof the Mason-Stothers theorem for an arbitrary field, and later on for a field with characteristic 0.

theorem *Mason_Stothers:*
  fixes *A B C :: "'a :: {factorial_ring_gcd, field} poly"*
  assumes *nz: "A ≠0" "B ≠0" "C ≠0" "∃p ∈{A,B,C}. pderiv p ≠0"*
    and sum: "A + B + C = 0"
    and coprime: "Gcd {A, B, C} = 1"
  shows *"Max {degree A, degree B, degree C} < degree (radical (A ∗ B ∗ C))"*

Now for an field of characteristic 0 the assumption *"∃p ∈{A,B,C}. pderiv p ≠0"* becomes *"∃p ∈{A,B,C}. degree p ≠0"* (which is equivalent to A, B, C not all constant). We note that by defining the gcd and the maximum for sets, the so called big-operators, we could obtain a similar clean statement of the theorem.

6. Discussion

In this chapter I discuss the usability of Lean for mathematicians and improvements that can be made on my formalization.

6.1. Usability

Setting Up and Running Lean

I had problems with the Lean package manager `leanpkg` under Windows. To circumvent this problem I installed a virtualized version of Ubuntu (Linux) on which I installed Lean. One cannot expect that a mathematician is an expert in solving software issues, hence the set-up should be smooth.

The evaluation speed of Lean was fast enough in small files, but at longer files it could slow the user down. Lean evaluates files from top to bottom. If a change is made in the file all lemmas after the change are re-evaluated. In a file with around 1500 lines the evaluation from top to bottom usually took longer than a minute. This was slowing me down as a user because I had to wait to see my current goal state. In addition there seemed to be a bug, so that sometimes when a change was made half way the file, it re-evaluated the entire file. The evaluation speed depends on the setup; I worked on a laptop, with Lean running on a virtualized operating system.

Learning to Formalize Using Lean

The step from reading the standard Lean introductory text and doing the exercises to performing a serious formalization proved challenging. The first part of learning to formalize is getting to know a proof assistant. For Lean there is an introductory text *Theorem proving in Lean* by Avigad et al. (2018b). This book gives an overview of formalization and the possibilities of Lean. Each chapter comes with exercises. Albeit this book gives a good overview, I experienced a big gap between, on the one hand, reading this book and doing the exercises and, on the other hand, performing a serious formalization. I suggests adding a chapter that aims to bridge this gap. This chapter should include the following subjects: the process of formalizing, Mathlib (the library), coding conventions, the channels for help from fellow formalizers.

6.1.1. Automation

Tactic documentation Lean has several useful tactics that are not well-documented, i.e. they were not described in the standard texts: *Theorem Proving in Lean* (Avigad

et al., 2018b) and *The Lean Reference Manual* (Avigad et al., 2018a). The rewrite tactic has the possibility to rewrite at a specific occurrence in an expression (using `occs := ...`), which is a common use-case, but was not discussed in the documentation. The important tactic `rcases`, which is used for existential elimination in tactic proofs, was never mentioned in the documentation. The combination tactics `rwa` and `rsimp`, which combine a rewrite or simplification with a finish by assumption, were also not documented.

Simple lemmas on natural numbers During the formalization I occasionally had to prove trivial facts about the natural numbers, these are prime candidates for future automation. As an example of such as a trivial lemma:

```
private lemma eq_zero_or_exists_eq_succ_succ_of_ne_one
{n : ℕ} (h : n ≠ 1) :
  n = 0 ∨ ∃ m, n = nat.succ (nat.succ m) :=
begin
  by_cases h2 : n = 0,
    {simp *},
    {
      rcases (nat.exists_eq_succ_of_ne_zero h2) with ⟨m, hm⟩,
      subst hm,
      simp * at *,
      rw succ_eq_succ_iff_eq at h,
      rcases (nat.exists_eq_succ_of_ne_zero h) with ⟨s, hs⟩,
      subst hs,
      exact ⟨s, rfl⟩,
    }
end
```

We see that the proof takes several lines. Lemmas as these would be solved by a *linear arithmetic* tactic, but this is not (yet) available in Lean.

6.1.2. Library

This formalization made heavy use of the Mathlib library. To name some subjects: the natural number (`nat`), finite sets (`finset`), finite multisets (`multiset`), finite support functions (`finsupp`), units (`unit`), rings (`ring`), fields (`field`), integral domains (`integral_domain`), lattices (`lattice`). Without these previously formalized notions, the duration of my formalization would have increased dramatically. I did extend most of these structures with lemmas that were needed for my formalization.

Data structures In this formalizations I occasionally had to prove lemmas about basic data structures, these proofs were often laborious. As an example a lemma on finite sets:

```
lemma finset.sum_ite_general''
  {α : Type u} {β : Type v} [decidable_eq α]
  [add_comm_monoid β] {f g : α → β} (s : finset α)
  (p : α → Prop) [decidable_pred p] :
    s.sum (λ z, if p z then f z else g z) =
      (s.filter p).sum f + (s.filter $ λx, ¬ p x).sum g :=
        proof
```

Here we sum an "if then else" function over a finite set. The proof took 28 lines. Proving and using lemmas on basic data structures is needed in any formalization. However a mathematician might not expect this, nor might he or she want to be bothered with developing basic data structures.

6.1.3. Lower Level Mathematics

In this formalization a great deal of lower level mathematics was formalized, namely: the associated quotient. When I wanted to prove that UFDs have a gcd, I used the quotient with respect to the associated relation. This structure is often used implicit by mathematicians, but is now made explicit. This is an example of where proof formalization reveals the hidden and implicit structure of mathematics, and hence could contribute to better understanding. Yet it is not all positive, because a mathematician would regard this associated quotient as trivial, and might not want to spend time to develop this structure.

6.2. Future Work

Besides the missing proof that was described in Section 4.2.2, there are also improvements that can be made on the parts that are already functional. In this section I point out several possible improvements.

Long proofs The formalization contains several lemmas with long proofs, these lengthy proofs could be due to inefficient implementation. I experienced that the length of my proofs decreased substantially as I learned more of Lean. As an estimate, the average length of my proofs is reduced by a factor 2 to 3 at the end of project, compared to the beginning of the project. When Mason-Stothers was formalized, I performed a clean-up of approximately $\frac{2}{3}$ of the code, where I reduced the length of the proofs using my improved skills. As I did not cover all the code, some of the longer proofs can potentially be shortened. At current the longest proofs are about the polynomial normal form, and the radical. I suggest performing a further clean-up of the code. This will lead to better insights on which proofs are inherently lengthy (and possibly need restructuring).

The is-unit predicate Currently the `is_unit` predicate is defined as

```
def is_unit (a : α) : Prop := ∃b : units α, a = b
```

Note that there is an implicit coercion here for `b`. Where `units` is the group of units of a type. It might give cleaner and shorter proofs to define

```
def is_unit (a : α) : Prop :=
  ∃ b : α, a * b = 1 ∧ b * a = 1
```

The reason to have it defined using the already existing `units` type, was because there were already several lemmas for this type.

Computational definitions The current definition of UFDs could be replaced by a computational definition. For UFDs an option would be to define a function

```
factorization: α → multiset α.
```

A computational approach leads to content that can be executed as a computer program, and we do not lose information due to existential quantifiers. For example, for polynomials over a field, which are a UFD, we could implement `factorization` using the polynomial normal form, which then gives more information than using the axiom of choice. In order to give such an implementation of `factorization` we need the algorithm that groups the leading coefficients of the irreducible factors; this algorithm can possibly be extracted from the lemma that proved the existence of the normal form. For the integers, the `factorization` function could be implemented to only return positive factors.

Fine structuring the algebraic structure graph At current the algebraic structure graph is: fields $\subset$ unique factorization domains $\subset$ integral domains $\subset$ rings. Which can be made finer: fields $\subset$ Euclidean domains $\subset$ principal ideal domains $\subset$ unique factorization domains $\subset$ Bezout domain $\subset$ gcd domain $\subset$ integral domain $\subset$ rings. In particular Euclidean-domains and Principal Ideal domains are very common ring types. And the gcd domain would be convenient to introduce because several lemmas proven for our formalization would already be valid in a gcd domain.

Scalar multiplication on polynomials For functions with finite support a scalar multiplication was defined. However this scalar multiplication is not used in the polynomial library developed here. Instead we always used polynomial multiplication with the constant polynomial (`C a`). Using the scalar multiplication could possibly reduce the size of some of the proofs.

Naming Several names I introduced should be reconsidered. I called the type for the associated quotient: `quot`, which is too general. The lifting of the irreducible predicate to the associated quotient I called: `irred`, however this could simply have been `irreducible`. The lifting of the factorization property to the associated quotient I called: `to_multiset`, this name could be misinterpreted as a coercion.

7. Conclusion

Proof assistants and mathematical formalization have several advantages, but are seldomly used by mathematicians, due to lack of automation, libraries and expertise. Lean is a new proof assistant, developed towards use by mathematicians. I performed a case-study, were I formalized the Mason-Stothers theorem in Lean. Lean proved suited for this task, and the single proof in the mathematical preliminary that is still missing from the formalization is merely a result of lack in time. In developing the proof, I had to formalize a great deal of background mathematics, about 4/5 of the formalization consisted of background. I developed a reusable library for polynomials and unique factorization domains. The formal proof of the Mason-Stothers closely resembles the informal proof. The most interesting proof of the formalization was that a gcd exists in a UFD, which required a substantial formalization of lower level mathematics (the associated quotient). I compared the main components of my formalization (polynomials, UFDs, and Mason-Stothers) to an independent formalization of these subjects in Isabelle. This let to ideas on possible improvements of my formalization, and showed that Lean performed well in some areas (the direct use of quotients structures, and having no need for explicit carrier sets). Regarding the usability of Lean, I discussed several possible improvements on the documentation. The automation in Lean has room for improvements. I used the library a lot during the formalization, but I had to extend it at times. Future work would start with filling the gap in the mathematical preliminary, followed by a further clean-up of the code.

Bibliography

Avigad J, Ebner G, Ullrich S (2018a) The Lean reference manual. URL https://leanprover.github.io/reference/lean_reference.pdf

Avigad J, de Moura L, Kong S (2018b) Theorem proving in Lean. URL https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf

Ballarin C, Hohe FK, Paulson LC (2017) The Isabelle/HOL algebra library. URL http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/HOL-Algebra/document.pdf

Castelvecchi D (2015) The impenetrable proof. Nature 526

Church A (1932) A set of postulates for the foundation of logic. Annals of Mathematics 33(2):346–366, URL http://www.jstor.org/stable/1968337

Dahmen S (2017) Number theory course notes

Dummit DS, Foote RM (2004) Abstract algebra, vol 3. Wiley Hoboken

Eberl M (2017) The Mason–Stother's theorem. Archive of Formal Proofs URL http://isa-afp.org/entries/Mason_Stothers.html

Geuvers H (2009) Proof assistants: History, ideas and future. Sadhana 34(1)

Hölzl J (2017) Polynomials in Lean, URL https://github.com/johoelzl/mason-stother/blob/183513ab78ac6f4b827371ac636150a399322fb5/poly.lean

Howard WA (1980) The formulae-as-types notion of construction, in J Seldin, R Hindley (eds), to HB Curry: essays on combinatory logic, lambda calculus and formalism. (original manuscript from 1969)

Huffman B, Ballarin C, Chaieb A, Haftmann F (2016) Polynomial. Archive of Formal Proofs URL https://isabelle.in.tum.de/library/HOL/HOL-Computational_Algebra/Polynomial.html

Lemmermeyer F (2005) Algebraic geometry (lecture notes). URL http://www.fen.bilkent.edu.tr/~franz/ag05/ag-02.pdf

Mahboubi A (2018) The Mason-Stother's theorem in Coq, unpublished

Mason RC (1984) Diophantine equations over function fields, vol 96. Cambridge University Press

de Moura L, Kong S, Avigad J, Van Doorn F, von Raumer J (2015) The Lean theorem prover (system description). In: International Conference on Automated Deduction, Springer, pp 378–388

Snyder N (2000) An alternate proof of Mason's theorem. Elem Math 55(3):93–94, DOI 10.1007/s000170050074, URL http://dx.doi.org/10.1007/s000170050074

Stothers WW (1981) Polynomial identities and hauptmoduln. The Quarterly Journal of Mathematics 32(3):349–370, DOI 10.1093/qmath/32.3.349, URL http://dx.doi.org/10.1093/qmath/32.3.349, /oup/backfile/content_public/journal/qjmath/32/3/10.1093/qmath/32.3.349/2/32-3-349.pdf

Thiemann R, Yamada A (2016) Polynomial factorization. Archive of Formal Proofs URL https://www.isa-afp.org/browser_info/devel/AFP/Polynomial_Factorization/Unique_Factorization_Domain.html

Wagemaker J, Hölzl J (2018) The Mason-Stothers theorem in Lean, URL https://github.com/johoelzl/mason-stother/

A. Mathematical Definitions for Rings

We follow the definition from Dummit and Foote (2004).

1. A *ring* R is a tuple $(S, +, \times)$, where S is a set called the *carrier* set, and $+$ and $\times$ are two binary operations on S called addition and multiplication. They satisfy the following axioms:
   a) $(S, +)$ is a *commutative* group.
   b) $\times$ is associative: $(a \times b) \times c = a \times (b \times c)$ for all $a, b, c \in S$.
   c) the *distributive laws* hold in R: for all $a, b, c \in S$
   $(a + b) \times c = (a \times c) + (b \times c)$ and $a \times (b + c) = (a \times b) + (a \times c)$.
2. The ring R is *commutative* if multiplication is commutative.
3. The ring R is said to have an *identity* (or *contain* a 1) if there is an element $1 \in S$ with $1 \times a = a \times 1$ for all $a \in S$.

Instead of saying the carrier set S of R, we will simply say R.

B. Mason-Stothers Theorem in Lean

```
theorem Mason_Stothers_special [field β]
  (h_char : characteristic_zero β)
  (a b c : polynomial β)
  (ha : a ≠ 0)
  (hb : b ≠ 0)
  (hc : c ≠ 0)
  (h_coprime_ab : coprime a b)
  (h_coprime_bc : coprime b c)
  (h_coprime_ca : coprime c a)
  (h_add : a + b = c)
  (h_constant :
     ¬(is_constant a ∧ is_constant b ∧ is_constant c)) :
  degree c < degree (rad (a*b*c)) :=

begin
  have h_der_not_all_zero
     : ¬(d[a] = 0 ∧ d[b] = 0 ∧ d[c] = 0),
  {
    rw [derivative_eq_zero_iff_is_constant h_char,
      derivative_eq_zero_iff_is_constant h_char,
      derivative_eq_zero_iff_is_constant h_char],
    exact h_constant,
  },
  have h_der : d[a] + d[b] = d[c],
  {
    rw [←h_add, derivative_add],
  },
  have h_wron : d[a] * b - a * d[b] = d[a] * c - a * d[c],
    from  h_wron a b c h_add h_der,
  have h_dvd_wron_a : gcd a d[a] ∣ d[a] * b - a * d[b],
    from h_dvd_wron_a a b c,
  have h_dvd_wron_b : gcd b d[b] ∣ d[a] * b - a * d[b],
    from h_dvd_wron_b a b c,
  have h_dvd_wron_c : gcd c d[c] ∣ d[a] * b - a * d[b],
    from h_dvd_wron_c a b c h_wron,
```

```
have h_gcds_dvd : (gcd a d[a]) * (gcd b d[b]) *
  (gcd c d[c]) ∣ d[a] * b - a * d[b],
  from h_gcds_dvd a b c h_coprime_ab h_coprime_bc
      h_coprime_ca h_dvd_wron_a h_dvd_wron_b h_dvd_wron_c,
have h_wron_ne_zero : d[a] * b - a * d[b] ≠ 0,
  from h_wron_ne_zero a b c h_coprime_ab
    h_coprime_ca h_der_not_all_zero h_wron,
have h_deg_add :
  degree (gcd a d[a] * gcd b d[b] * gcd c d[c]) =
    degree (gcd a d[a]) + degree (gcd b d[b]) +
      degree (gcd c d[c]),
  from h_deg_add a b c h_wron_ne_zero h_gcds_dvd,
have h_deg_add_le : degree (gcd a d[a]) +
  degree (gcd b d[b]) +
    degree (gcd c d[c]) ≤
      degree a + degree b - 1,
{
  rw [←h_deg_add],
  have h1 : degree (gcd a d[a] * gcd b d[b] * gcd c d[c]) ≤
    degree (d[a] * b - a * d[b]),
    from degree_dvd h_gcds_dvd h_wron_ne_zero,
  exact nat.le_trans h1 (degree_wron_le),
},
have h_deg_c_le_1 :
  degree c ≤ (degree a - degree (gcd a d[a])) +
    (degree b - degree (gcd b d[b])) +
      (degree c - degree (gcd c d[c])) - 1,
  from rw_aux_1 a b c h_add h_constant h_deg_add_le,
have h_le_rad : degree a - degree (gcd a d[a]) +
    (degree b - degree (gcd b d[b])) +
      (degree c - degree (gcd c d[c])) - 1 ≤
degree (rad (a * b * c)) - 1,
  from rw_aux_2 ha hb hc h_coprime_ab
    h_coprime_bc h_coprime_ca,
have h_ms : degree c ≤ degree (rad (a*b*c)) - 1,
  from nat.le_trans h_deg_c_le_1 h_le_rad,
have h_eq : degree (rad (a*b*c)) - 1 + 1 =
  degree (rad (a*b*c)),
{
  have h_pos : degree (rad (a*b*c)) > 0,
    from degree_rad_pos a b c ha hb hc h_constant,
  apply nat.succ_pred_eq_of_pos h_pos,
},
exact show degree c < degree (rad (a*b*c)),
```

```
    from calc degree c + 1 ≤
      degree (rad (a*b*c) ) - 1 + 1 : by simp [h_ms]
      ... = degree (rad (a*b*c)) : h_eq
end
```