Bachelor Computer Science

Bachelor thesis

An Implementation of Buchberger's Algorithm in Lean

Markos Dermitzakis

April 24, 2019

Supervisor: Dr. Jasmin C. Blanchette

Daily Supervisor: Dr. Robert Y. Lewis

Second Examiner: Dr. Jasmin C. Blanchette

Department of Computer Science
Faculty of Sciences

Abstract

Interactive proof assistants are a powerful tool in the hands of mathematicians, students and anyone interested in constructing proofs and verify their correctness. Lean is a new, open source programming language and tool for formalizing mathematical proofs. In this thesis I use Lean to implement Buchberger's Algorithm and the Buchberger criterion. To develop the implementation I used preexisting definitions from Lean's library *mathlib* and added some backgroung mathematics including the S-polynomial and Long Division for multivariate polynomials.

Title: An Implementation of Buchberger's Algorithm in Lean
Author: Markos Dermitzakis, markos.dermitzakis@hotmail.com
Supervisor: Dr. Jasmin C. Blanchette
Daily supervisor: Dr. Robert Y. Lewis
Second examiner: Dr. Jasmin C. Blanchette
Date: April 24, 2019

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

Contents

1 Introduction

The notion of Gröbner bases and the fundamental results of Gröbner basis theory were introduced in 1965, together with an algorithm to compute them, by Bruno Buchberger in his Ph.D. thesis [1], naming them in honor of W. Gröbner (1899–1980), Buchberger's thesis adviser. Gröbner bases became an important theoretical building block of ring theory. The origin of Gröbner basis theory goes back to solving some theoretical problems concerning the ideals in polynomial rings, as well as solving polynomial systems of equations. Subsequently, Gröbner bases found further use in a number of applications in commutative algebra (intersection of ideals, envelopes, Hilbert dimension), other areas of mathematics (geometric theorem proving, chromatic numbers of graphs), applications in engineering (coding theory, general inverse kinematics problem in robotics, software engineering, formal verification of digital circuits) and many other areas, after reformulating a problem as set of multivariate polynomial equations and exploiting the properties of the Gröbner basis [2][3].

We can use computer based tools to state theorems, build formal proofs and verify their correctness, with several advantages such as attention to details and no errors. Lean is a modern system for building mathematical libraries and stating and proving mathematical theorems. Although I have almost no prior experience with proof assistant tools, in this bachelor thesis I used the Lean proof assistant to implement the Buchberger Criterion for identifying whether a basis for a polynomial ideal is a Gröbner basis and the Buchberger algorithm for computing Gröbner bases for polynomial ideals.

This thesis is divided in 4 chapters. Chapter 2 introduces the mathematical concepts for Gröbner bases and the mathematical methods used for the computation. Chapter 3 describes the implementation of Buchberger's criterion and algorithm and chapter 4 presents some conclusive thoughts about possible future work and the usability of Lean.

2 Background

2.1 Mathematical preliminaries

For this project, the following structures have been defined as in [4] and [5]:

Rings A ring (R) is a nonempty set equipped with two operations that satisfy the following axioms:

- R is closed under addition and multiplication:

  if $a, b \in R$ then $a + b \in R$ and $a \cdot b \in R$

- Addition and multiplication are associative:

  if $a, b \in R$ then $(a + b) + c = a + (b + c)$ and $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

- Addition is commutative:

  if $a, b \in R$ then $a + b = b + a$

- There exists an additive identity (or 0 element) in R, such that for all $a \in R$:

  $a + 0_R = a = 0_R + a$

- There exists a multiplicative identity 1 in R, such that for all $a \in R$:

  $a \cdot 1 = a$ and $1 \cdot a = a$

- For all $a \in R$, there exists an additive inverse x, such that: $a + x = 0_R$

- The distributive laws of multiplication hold in R:

  if $a, b, c \in R$ then $a \cdot (b + c) = a \cdot b + a \cdot c$ and $(a + b) \cdot c = a \cdot c + b \cdot c$

Commutative Rings A commutative ring is a ring where multiplication is also commutative.

Monomials Given n variables $x_1, ..., x_n$ and $a_1, ..., a_n$ nonnegative integers, a monomial in $x_1, ..., x_n$, is a product of the form:

$$x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdot \ldots \cdot x_n^{\alpha_n}$$

Polynomials A polynomial p in $x_1, ..., x_n$ with coefficients in k is a finite linear combination of monomials of the form:

$$\sum_{\alpha} a_\alpha \cdot m_\alpha, a_\alpha \in k,$$

where m_α is a monomial in $x_1, ..., x_n$. The set of all polynomials in $x_1, ..., x_n$ with coefficients in k is denoted $k[x_1, ..., x_n]$.

Coefficient and term of a monomial Given a polynomial $p = \sum_\alpha a_\alpha \cdot x^\alpha$:

- a_α is the coefficient of the monomial x^α and
- if $a_\alpha \neq 0$, then $a_\alpha \cdot x^\alpha$ is a term of p.

Ideals An ideal (I) is a nonempty subset of a ring R that is closed under addition and multiplication with ring elements. That is:
if $a, b \in I$ then $(a + b) \in I$ and
if $a \in I$ and $r \in R$ then $a \cdot r \in I$ and $(r \cdot a) \in I$
Given polynomials $p_1, \ldots, p_s \in k[x_1..x_n]$ then $\langle p_1, \ldots, p_n \rangle$ is an ideal (I) of $k[x_1, ..., x_n]$. The set $\langle p_1, \ldots, p_n \rangle$ consists of all polynomial linear combinations and is called the ideal generated by $p_1, \ldots, p_s$.
We also call initial ideal of I, denoted $in(I)$, the ideal generated by the *leading terms* of all the polynomials in I. I define the *leading term* in 2.2.1.

Fields A field is a commutative ring in which each nonzero element has a multiplicative inverse. Equivalently, a field is a commutative ring (R) in which the only ideals are 0 and R itself.

Monomial Ideal A monomial ideal $I \subset k[x_1, ..., x_n]$ is a polynomial ideal where there is a subset $A \subset \mathbb{Z}^n_{\geq 0}$ such that I consists of all polynomials which are finite sums of the form $\sum_{\alpha \in A} h_\alpha x^\alpha$, where $h_\alpha \in k[x_1, ..., x_n]$.

Basis An ideal I is finitely generated if there exist $(p_1, ..., p_s) \in k[x_1, ..., x_n]$, such that $I = \langle p_1, ..., p_n \rangle$. Then $p_1, ..., p_s$, are a set of generators or a **basis** of I.

At this point, is important to highlight the fundamental property stated by the *Hilbert Basis Theorem*, that is:

Theorem 1 (Hilbert Basis Theorem) *Every ideal $I \subset k[x_1, ..., x_n]$ has a finite generating set. That is, $I = \langle p_1, ..., p_t \rangle$ for some $p_1, ..., p_t \in I$.*

By applying the Hilbert Basis Theorem to any ascending chain of ideals in $k[x_1, ..., x_n]$: $I_1 \subset I_2 \subset I_3 \subset ...$, we can state that the ascending chain will be stabilized after a finite number of steps, in the sense that all the ideals after that point in the chain will be equal. This statement is called **ascending chain condition** (ACC).
A given ideal may have several different bases, depending on the ordering of terms in polynomials.

Monomial orderings Given that polynomials are a sum of monomials, it is useful to be able to arrange their terms in an ascending or descending order, given some total order. Hence a monomial ordering on $k[x_1, ..., x_n]$ is any well-ordering relation on the set of monomials $x^\alpha, \alpha \in \mathbb{Z}^n_{\geq 0}$ which for all $\beta, \gamma \in \mathbb{Z}^n_{\geq 0}$ also satisfies the conditions:

- if $\alpha \neq \beta$, then either $x^\alpha \cdot x^\gamma < x^\beta \cdot x^\gamma$ or $x^\alpha \cdot x^\gamma > x^\beta \cdot x^\gamma$
- if $\alpha > \beta$, then $\alpha + \gamma > \beta + \gamma$

Given $\alpha = (\alpha_1, ..., \alpha_n)$ and $\beta = (\beta_1, ..., \beta_n) \in \mathbb{Z}^n_{\geq 0}$, there are several term orderings of interest in computational algebra. Among them:

- **Lexicographic** ("dictionary"): Then $\alpha >_{lex} \beta$ if the left-most nonzero entry of $\alpha - \beta$ is positive.
  e.g $(1, 2, 0) >_{lex} (0, 3, 4)$ and $(3, 2, 4) >_{lex} (3, 2, 1)$
- **Graded Lexicographic**: Sorting is initially performed by total degree ($|\alpha| = \alpha_1 + ... + \alpha_n$, $|\beta| = \beta_1 + ... + \beta_n$) and if $|\alpha| = |\beta|$ then using the lexicographic.
  e.g $(1, 2, 3) >_{grlex} (3, 2, 0)$ since $|(1, 2, 3)| > |(3, 2, 0)|$ and $(1, 2, 4) >_{grlex} (1, 1, 5)$ since $|(1, 2, 4)| = |(1, 1, 5)|$ but $(1, 2, 4) >_{lex} (1, 1, 5)$

Gröbner basis A Gröbner basis is a particular kind of generating set of an ideal I in a polynomial ring $k[x_1, .., x_n]$ over a field k. Given I, a finite subset G of I is a Gröbner basis, if the initial terms (terms with the largest monomial with respect to a monomial

order) of the elements in G suffice to generate the initial ideal.
Given I we can construct 2 monomial ideals associated with it: The initial ideal *in(I)* and the monomial ideal generated by the initial monomials of the generators that is $\langle in(p_1), \ldots, in(p_n)\rangle$. A finite set of polynomials $\{p_1, ..., p_s\} \subset I$ is a Gröbner basis of I if the initial ideal of I is generated by the leading terms of p_i, that is if $in(I) = \langle in(p_1), \ldots, in(p_n)\rangle$.
Moreover, the corollary below follows from 1:

Corollary 1.1 *Given a monomial order, every ideal I other than 0 has a Gröbner basis G and* $I = \langle G\rangle$.

2.2 The Buchberger Algorithm

2.2.1 Background

Given the importance of Gröbner bases, Buchberger in his 1965 Ph.D. thesis, came up with the first algorithm to compute them. His algorithm is characterized for relying on the operations of the S-polynomial of two polynomials and the long division for multivariate polynomials.

The S-Polynomial The S-polynomial makes use of the notions *Leading Monomial*, *Leading Coefficient*, *Leading Term* and *Least Common Multiple* of a set of monomials. Let $f, g \in k[x_1, ..., x_n]$ be nonzero polynomials and let a monomial order.

- **Leading Monomial (LM)**: LM(f) is the maximal monomial that appears in f, resulting from the monomial ordering.
- **Leading Coefficient (LC)**: LC(f) is the coefficient of the LM(f)
- **Leading Term (LT)**: LT(f) is LC(f) $\cdot LM(f)$
- **Least Common Multiple (LCM)**: Let $f = \sum_\alpha a_\alpha \cdot x^\alpha$. If $f, g \setminus \{0\}$ then the multidegree of f is defined as:

  $$multideg(f) = max(\alpha \in \mathbb{Z}^n_{\geq 0} : \alpha_\alpha \neq 0),$$

  where the maximum is taken with respect to the monomial order.
  If $multideg(f) = \alpha$ and $multideg(g) = \beta$, let $\gamma = (\gamma_1, ..., \gamma_d)$ where $\gamma_i = max(\alpha_i, \beta_i)$.

Let χ^γ be the LCM(LM(f), LM(g)). The S-polynomial operation aims to cancel out the leading terms of f and g and is defined as the combination:

$$S(f,g) = \frac{\chi^\gamma}{LT(f)} \cdot f - \frac{\chi^\gamma}{LT(g)} \cdot g$$

The Multivariate Division Algorithm The Multivariate Division Algorithm is an extension of the division algorithm for polynomials in one variable, allowing us to divide every $p \in k[x_1, ..., x_n]$ by a finite ordered sequence of polynomials $P = (p_1, ...p_s)$, where $P \in k[x_1, ..., x_n]$, and making possible to express p in the form $p = q_1 \cdot p_1 + ... + q_s \cdot p_s + r.$, where $q_i, r \in k[x_1, ..., x_n]$, and either the remainder $r = 0$ or is a linear combination with coefficients in k, of monomials, none of which is divisible by any of $LT(p_1), ..., LT(p_s)$. Nevertheless, while the division algorithm for $k[x]$ guarantees a unique and well defined output of a quotient *(q)* and remainder *(r)*, in the case of polynomials in $k[x_1, ..., x_n]$, the result of the algorithm is not unique, depending on the chosen monomial ordering and the order of the divisors during the operation. During the process, the leading term of p is repeatedly canceled by subtracting the appropriate multiple of one of the p_i. We write $\overline{p}^G$ for the remainder in the division of p by the ordered list of polynomials $G = \{g_1, ..., g_s\}$.

Theorem 2 (Buchberger Criterion) *Let I be a polynomial ideal. Then a basis $G = \{g_1, ..., g_t\}$ for I, given some monomial order, is a Gröbner basis for I, if and only if for all pairs $i \neq j$, the remainder on division of $S(g_i, g_j)$ by G is zero. That is if $\overline{S(g_i, g_j)}^G = 0$.*

This theorem enables us to verify whether a given basis is a Gröbner basis or not, by outlining Gröbner bases in terms of the S-polynomials. The criterion is the basis of the Buchberger's algorithm, which will produce the Gröbner bases for a nonzero ideal.

2.2.2 Theorem statement

Buchberger's algorithm exploits Corollary 1.1, which is that every ideal I other than 0, has a Gröbner basis and provides an efficient way for the computation. Combined, Buchberger's Criterion and algorithm, provide an algorithmic basis for the theory of Gröbner bases. These contributions of Buchberger are central to the development of the subject.

The steps are to start with a basis of the ideal - not necessarily a Gröbner basis - and compute S-polynomials which are reduced by the polynomials found so far. If an S-polynomial reduces to zero we proceed with the operation. If an S-polynomial has a

nonzero remainder, then we add this remainder to the current list of polynomials. We keep doing this until we end up with a Gröbner basis, which is ensured by Buchberger's criterion.

Buchberger Algorithm Let $I = \langle f_1, \ldots, f_s \rangle \neq 0$ be a polynomial ideal. Then a Gröbner basis for I can be constructed in a finite number of steps by the following algorithm:

Input: $F = (f_1, ..., f_s)$

Output: a Gröbner basis $G = (g_1, ..., g_t)$ for I, with $F \subset G$

$G := F$

REPEAT

$G' := G$

FOR each pair {p,q}, $p \neq q$ in G' DO

$S := \overline{S(p,q)}^{G'}$

IF $S \neq 0$ THEN $G := G \cup \{S\}$

UNTIL $G = G'$

At termination, $S := \overline{S(p,q)}^{G'} = 0$, for all $p, q \in G$, thus following from the Buchberger criterion that the result produced is a Gröbner basis.

This process ends after a finite number of steps. The fact the algorithm will terminate is determined by the ascending chain condition described in theorem 1. In particular, each iteration will increase the monomial ideal generated by the leading terms of the elements of G. Since $G_i \subseteq G_{i+1}$, each of the ideals will be a sub-ideal of the following iteration and by ACC this operation will eventually terminate.

3 Implementation

The realization of the project was performed in Lean (v.3.4.2) proof assistant. Lean is a modern tool developed at Microsoft Research and Carnegie Mellon University, which allows to write and to prove mathematical theories and supporting the formalization of mathematics. Lean aims to bridge the gap between interactive and automated theorem proving, by situating automated tools and methods in a framework that supports user interaction and the construction of fully specified axiomatic proofs. It makes use of its standard library, *mathlib*, which is constantly growing and the content enriched with further definitions, proofs, guidelines, processes and information. Combined with the power and flexibility provided by the strong proof automation, it is an important tool to perform accurate computations and reasoning in an efficient way.

In addition, Lean allows mathematical assertions and proofs in the language of dependent type theory as well, making it flexible and more expressive. For example, with the type, `Prop`, we can represent and state propositions, while by introducing constructors we can build new propositions from others. By defining the rules to be used and the proofs for a given proposition, we can add new rules and new proofs, extending the libraries and the ability for more complex projects.

Overall, the goal is to support both mathematical reasoning and reasoning about complex systems, and to verify claims in both domains. Lean also has mechanisms to serve as its own metaprogramming language, which means that one can implement automation and extend the functionality of Lean using Lean itself.

More complete documentation of the system can be found in the system's publications [6][7][8][9][10], documentation [11] and website [12].

In this chapter I discuss the process of implementation of Buchberger's algorithm and the necessary background mathematics added with the formalization of some predicates that describe its behavior.
My code can be found on GitHub at `https://github.com/MarkosDe/bb_alg.git`

3.1 Multiariate Polynomial and related notions

The multivariate polynomial had already been defined in *mathlib*, but in an inconvenient for computations way. A new definition was necessary along with the definitions of a proper monomial ordering. For this purpose, it was convenient to use the concept of multiset contained in *mathlib* library and defined as:

```
def {u} multiset (α : Type u) : Type u :=
   quotient (list.perm.setoid α)
```

The above results in a finite unordered set with duplicates of elements allowed and we are able to express a monomial e.g. x^2yz^3 in terms of a multiset as $\{x, x, y, z, z, z\}$.

3.1.1 Monomial order

At this point is fondamental to define a monomial ordering for a finite set of elements of type σ. The monomial order used is the lexicographic and was defined as:

```
variables {σ : Type*} [decidable_linear_order σ]

def multiset.lex (m1 m2 : multiset σ) : Prop :=
   m1.sort (≥) ≤ m2.sort (≥)
```

Now we can sort the elements of the multiset and the operation needs to be defined as decidable in order to be able to make use of it in building propositions and making computable functions. *mathlib* contains a decidable type class which can be used to infer a procedure that effectively determines whether or not the proposition is true. As a result, the type class supports such computational definitions when they are possible.

```
instance (m1 m2 : multiset σ) : decidable (multiset.lex m1 m2) :=
   by unfold multiset.lex; apply_instance
```

In order for the lexicographic order to be used as monomial ordering, it must satisfy the properties of total ordering, well ordering and respecting addition. The first two properties are showed with the definition:

```
def multiset.lex_is_total (σ) [decidable_linear_order σ]:
    decidable_linear_order (multiset σ) :=
{ le := multiset.lex,
  le_refl := λ m, le_refl _,
  le_trans := λ m1 m2 m3 h1 h2, le_trans h1 h2,
  le_antisymm := λ m1 m2 h1 h2, multiset.sort_ext.2 (le_antisymm h1 h2),
  le_total := λ m1 m2, le_total _ _,
  decidable_le := λ _ _, by apply_instance
}
```

The last property has only been assumed to be true, since:

for $x^\alpha > x^\beta$, then $x^\alpha + x^\gamma > x^\beta + x^\gamma$.

The lexicographic order defined will be the default monomial order used in this project, but if another order is defined, it is easy to substitute in.

3.1.2 The multivariate polynomial and decomposition

Once we can define monomials and we have a proper monomial ordering, we are able to provide a definition for the multivariate polynomial. For this definition, *mathlib* is used again for the previously formalized notions of commutative semirings and the subtype of finitely supported functions (`finsupp`). `finsupp` is defined as a type of functions from α to β, $(f : \alpha \to \beta)$, for which the set of elements in the domain that are mapped to a nonzero value is finite, (finite $\{a \mid f\ a \neq 0\}$).

The multivariate polynomial can now be defined as:

```
def my_mvpolynomial (σ : Type*) (α: Type*) [comm_semiring α]: Type* :=
  finsupp (multiset σ) α
```

The function returns the type of multivariate polynomials over α within the domain of a commutative semiring.

```
variables (α : Type u) (σ : Type*) [decidable_linear_order σ]
    [discrete_linear_ordered_field α]

instance : comm_ring (my_mvpolynomial σ α) :=
  finsupp.to_comm_ring
```

Monomials and one term polynomials have a coefficient and a definition is provided to extract it, making use of the library's `finsupp.to_fun`.

```
def my_mvpolynomial.coef (pol : my_mvpolynomial σ α) : multiset σ -> α :=
  pol.to_fun
```

With the definitions above and the *mathlib* library we can now decompose a multivariate polynomial and define the necessary notions that will be used for the computation of the s-polynomial and the long division.

The leading monomial is defined by the function below, which determines and returns the maximum monomial of a polynomial:

```
def my_mvpolynomial.leading_mon(pol: my_mvpolynomial σ α): option(multiset σ):=
  (finsupp.support pol).max
```

The function returns a type `option (multiset)`, that is, if there is a leading monomial then returns the maximum multiset and none otherwise.

The leading monomial is used in the definition computing the leading coefficient of our multivariate polynomial. A definition is:

```
def my_mvpolynomial.leading_coef (pol: my_mvpolynomial σ α): α :=
   match (my_mvpolynomial.leading_mon pol) with
   | some ms := pol.coef ms
   | none := 0
end
```

With the definition above, the function takes as parameter a multivariate polynomial and if it has a leading monomial, the function will return this monomial's type α coefficient and 0 otherwise.

Since we have defined the leading monomial and leading coefficient of a multivariate polynomial,we are able to also define the leading term, by simply mapping the leading coefficient to the leading monomial. Thus, given a polynomial *pol*, the function below returns *pol*'s leading monomial, of type *my_mvpolynomial* and 0 if *pol* is the *zero polynomial* (the constant polynomial whose coefficients are all equal to 0).

```
def my_mvpolynomial.l_t (pol: my_mvpolynomial σ α ) : my_mvpolynomial σ α :=
   finsupp.single (my_mvpolynomial.leading_mon pol).iget
    (my_mvpolynomial.leading_coef pol)
```

At this point it would be useful to be able to represent in Lean the multivariate polynomial and the results of the operations defined until now. Since the definition of multivariate polynomial was not present in the library, a possible representation has been provided and defined:

```
def my_mvpolynomial.repr [has_repr σ] [has_repr α] (p : my_mvpolynomial σ α) :
    string :=
   if p = 0 then "0" else ((finsupp.support p).sort multiset.lex).foldr
   (λ ms s, let coef := p.to_fun ms in
   s ++ (if (coef ≥ 0 ∧ s.length > 0) then " +" ++ repr coef else " " ++ repr
    coef) ++ repr ms ) ""

instance [has_repr σ] [has_repr α] : has_repr (my_mvpolynomial σ α) :=
   ⟨my_mvpolynomial.repr⟩
```

As an example, we can define a polynomial (*example_pol*), which for the sake of simplicity has variables inferred by $\mathbb{N}$, $(1, 2, 3)$ and their coefficients in $\mathbb{R}$. The *mathlib* library contains the finitely supported function constructor `finsupp.single a b`, which enables to map a value b of type β to a of type α, `finsupp α β`, and 0 otherwise. e.g:

```
def example_pol: my_mvpolynomial ℕ ℚ :=
   finsupp.single {3,1} (1) -
   finsupp.single {2,2} 1
```

Using the defined representation, the polynomial above and the results of the operations defined are represented as:

```
#eval example_pol : 1{1, 3} -1{2, 2}
#eval my_mvpolynomial.leading_coef example_pol: 1
#eval my_mvpolynomial.leading_mon example_pol: (some {1, 3})
#eval my_mvpolynomial.l_t example_pol: 1{1, 3}
```

3.2 The S-polynomial

The operations described in this part were defined in order to implement the fundamental operation of the S-polynomial. As described in part 2.2.1, for the computation of the S-polynomial it is essential to further define the operations of Least Common Multiple (LCM) between a set of monomials and the division of monomials.

Given that monomials are being represented with multisets, we can compute the LCM between them by constructing a new monomial, containing the LCM for each of their variables. To help us with this operation I have defined the function *add_repeats*, which adds to a multiset, in our case an empty multiset, a desired number of repetitions for a given element.

```
def multiset.add_repeats (ms: multiset σ) (aa : σ) (n: nat) : multiset σ :=
  (multiset.repeat aa n) + ms

e.g.: #eval multiset.add_repeats ({} : multiset ℕ) 2 5
```

will produce {2, 2, 2, 2, 2}

The LCM between a set of monomials can now be defined as:

```
def multiset.mon_LCM (m1 m2 : multiset σ) : multiset σ :=
  ((m1.to_finset) ∪ (m2.to_finset)).fold multiset.add ({})
  (λ x, multiset.add_repeats ({} : multiset σ) x (max (multiset.count x m1)
    (multiset.count x m2)))
```

In this way, the occurrences for every variable present in the set of monomials are counted, and their max is copied to a new multiset, thus constructing their LCM.

```
e.g.: #eval multiset.mon_LCM ({3,3,2} : multiset ℕ) ({2,2,8} : multiset ℕ)
```

gives {2, 2, 3, 3, 8}.

The operation of division between monomials, given our definition in terms of multisets, is equivalent to the conveniently defined operation of subtraction between them and present in *mathlib*:
e.g. given the monomials $ms_1 : x^2y^2z^3$ and $ms_2 : xyz$, then $\frac{ms_1}{ms_2} = xyz^2$.

In terms of our definitions up to now, $ms_1 : \{x, x, y, y, z, z, z\}$, $ms_2 : \{x, y, z\}$ and $ms_1 - ms_2 = \{x, y, z, z\}$ Therefore we do not need further definitions to implement the division between monomials.

As discussed in 2.2.1, the S-polynomial is defined as:

$$S(f, g) = \frac{\chi^\gamma}{LT(pol_1)} \cdot pol_1 - \frac{\chi^\gamma}{LT(pol_2)} \cdot pol_2$$

At this point we have all the essential operations for its computation between two polynomials and has been decomposed in the steps below:
Given two polynomials pol_1 and pol_2, let LM the leading monomial of a multivariate polynomial, LT the leading term and χ^γ the $LCM(LM(pol_1), LM(pol_2))$. The computation of $\frac{\chi^\gamma}{LM(pol_1)}$ is first perfomed, producing a new monomial and defined as:

```
def s_monomial_l (pol1 pol2: my_mvpolynomial σ α): multiset σ :=
  let lmp1 := (my_mvpolynomial.leading_mon pol1).iget in
  (multiset.mon_LCM lmp1 (my_mvpolynomial.leading_mon pol2).iget) - lmp1
```

The left hand side of the S-polynomial,

$$\frac{\chi^\gamma}{LT(pol_1)} \cdot pol_1,$$

can now be computed, by taking the monomial resulting from the previous operation, computing its coefficient and multiplying it with pol_1.

```
def s_polynomial_l (pol1 pol2: my_mvpolynomial σ α) : my_mvpolynomial σ α :=
  (finsupp.single (s_monomial_l pol1 pol2) (1 / (my_mvpolynomial.leading_coef
   pol1))) * pol1
```

We repeat the last operations to obtain the right hand side of the S-polynomial, first for the computation of:

$$\frac{\chi^\gamma}{LM(pol_2)}$$

```
def s_monomial_r (pol1 pol2: my_mvpolynomial σ α): multiset σ :=
   let lmp2 := (my_mvpolynomial.leading_mon pol2).iget in
   (multiset.mon_LCM (my_mvpolynomial.leading_mon pol1).iget lmp2) - lmp2
```

and lastly for the the right hand side:

$$\frac{\chi^\gamma}{LT(pol_2)} \cdot pol_2$$

```
def s_polynomial_r (pol1 pol2: my_mvpolynomial σ α) : my_mvpolynomial σ α :=
   (finsupp.single (s_monomial_r pol1 pol2) (1 / (my_mvpolynomial.leading_coef
    pol2))) * pol2
```

Finally, the S-polynomial is computed by simply subtracting the 2 resulting polynomials:

```
def s_polynomial (pol1 pol2: my_mvpolynomial σ α) : my_mvpolynomial σ α :=
   (s_polynomial_l pol1 pol2) - (s_polynomial_r pol1 pol2)
```

3.3 The Long Division

The purpose of the long division algorithm is to cancel the leading term of a multivariate polynomial $f \in k[x_1, ..., x_n]$ (with respect to a fixed monomial order), by multiplying some $f_i \in k[x_1, ..., x_n]$ by an appropriate monomial and subtracting. In order to achieve this, we need to be able to evaluate whether a $LM(f_i)$ divides any monomial of f.

For this purpose a function which performs this operation is needed and was defined as:

```
def divide_witness (pol_d pol_n : my_mvpolynomial σ α) : option (multiset σ) :=
   let lmp1 := (my_mvpolynomial.leading_mon pol_d).iget in
   list.find (λ s2, ∀ s ∈ lmp1, (lmp1).count s ≤ (s2: multiset σ).count s)
    ((finsupp.support pol_n).sort multiset.lex)
```

A monomial ms_d divides another monomial ms_n, if the number of each element contained in ms_d is $\leq$ than the number of each element contained in ms_d. This function iterates through the elements of the leading monomial of f_i and outputs the first monomial of f,

if any, type *multiset* σ, that divides and none otherwise.
The resulting monomial can now be mapped to a coefficient:

```
def divide_witness_pol(pol_d pol_n:my_mvpolynomial σ α) : my_mvpolynomial σ α:=
   match divide_witness pol_d pol_n with
   | some ms := finsupp.single ms (pol_n.to_fun ms)
   | none := 0
end
```

Using the operations defined above, we can now start implementing a division algorithm and decompose it in several different steps. Let n the dividend, d the divisor and r the remainder, then the division algorithm is described in what follows:

An operation needs to be defined allowing to divide the leading terms between the output of the `divide_witness` (the monomial of f divided by the leading monomial of f_i) and $LT(f_i)$.

```
def find_pol_t (pol_r pol_d: my_mvpolynomial σ α) : my_mvpolynomial σ α :=
   let dw := (divide_witness pol_d pol_r).iget in
   finsupp.single (dw - (my_mvpolynomial.leading_mon pol_d).iget)
   ((divide_witness_pol pol_d pol_r).coef dw / my_mvpolynomial.leading_coef
    pol_d)
```

With the operation above, the monomial resulting from the `divide_witness` function is divided with the $LM(f_i)$ and mapped to the coefficient resulting from the division of their respective coefficients, resulting into a new polynomial.

The quotient (q) resulting from the division between the polynomials is not essential for the implementation of Buchberger's algorithm, as we are only interested in the remainder of the operation. Nevertheless, it can be useful for future use and was defined as:

```
def find_new_q (pol_r pol_d pol_q: my_mvpolynomial σ α) : my_mvpolynomial σ α :=
   (pol_q + find_pol_t pol_r pol_d)
```

Another function has been defined repeatedly aiming to remove the leading term of the remainder.

```
def find_new_r (pol_r pol_d: my_mvpolynomial σ α) : my_mvpolynomial σ α :=
   pol_r - (find_pol_t pol_r pol_d) * pol_d
```

The remainder is updated, while always preserving the property that $n = d \cdot q + r$.

With the definitions above, we have all operations necessary for the implementation of the Long Division algorithm. Nevertheless, it would be appropriate to also be able to

prove that the algorithm to be defined also terminates. For this reason, a well founded relation must be constructed that will be used to show that its recursive application is decreasing and will finally end.

```
def wf_rel : (Σ' n : my_mvpolynomial σ α, list(my_mvpolynomial σ α)) →
(Σ' n : my_mvpolynomial σ α , list (my_mvpolynomial σ α)) → Prop :=
   λ m1 m2, ((m2.1.leading_mon).iget).sort (≥) < ((m1.1.leading_mon).iget).sort
    (≥)
```

The relation used is the decreasing leading monomial resulting after each iteration.

```
lemma wf_rel_wf : well_founded (wf_rel α σ) :=
   sorry
```

The relation created has not been proved to be a well founded relation, but it has been assumed to be true from section 2.2.1.

Finally, using the definitions introduced, we can define a well founded Long Division algorithm, allowing us to divide $f \in k[x_1, ..., x_n]$ by $f_1, ..., f_s \in k[x_1, ..., x_n]$.

```
def long_div: my_mvpolynomial σ α -> list (my_mvpolynomial σ α) ->
    my_mvpolynomial σ α
| pol_n (h::t) :=
    have h1 : wf_rel α σ ⟨find_new_r pol_n h, h::t⟩ ⟨pol_n, h::t⟩, from sorry,
    have h2 : wf_rel α σ ⟨pol_n, t⟩ ⟨pol_n, h::t⟩, from sorry,
    if (divide_witness h pol_n) = none then long_div pol_n t
    else long_div (find_new_r pol_n h) (h::t)
| pol_n [] := pol_n
using_well_founded {rel_tac := λ α σ, `[exact ⟨wf_rel α σ, wf_rel_wf α σ⟩],
    dec_tac := `[assumption]}
```

The algorithm proceeds with the division of a polynomial `pol_n` with an ordered list of polynomials, by repeatedly testing whether some $LT(f_i)$ divides $LT(pol_n)$ and if possible divide until no longer possible. Given that after each iteration the leading monomial of the updated remainder will be smaller, if not zero, by making use of the well founded relation we are able to show that the algorithm will also eventually terminate.

3.4 The Buchberger Criterion

We can exploit Theorem 2 to implement an operation, testing whether $G = \{g_1, ..., g_t\}$ for an ideal I is a Gröbner basis or not.

```
def bb_criterion (basis: list (my_mvpolynomial σ α)) : bool :=
  ∀ x ∈ basis, ∀ y ∈ basis, x ≠ y -> (long_div (s_polynomial x y) (basis) = 0)
```

The function above is a boolean type, testing if for all pairs $i \neq j$, the remainder of the long division operation $S(g_i, g_j)$ by G is zero. That is if $\overline{S(g_i, g_j)}^G = 0$. If the last holds, then the given basis is a Gröbner basis and the function will return *true*, otherwise *false*. As an example we can construct 2 polynomials f_1, f_2:

```
def f1: my_mvpolynomial ℕ ℚ := finsupp.single {3,3,2} 1 - finsupp.single {} 1
def f2: my_mvpolynomial ℕ ℚ := finsupp.single {3,2,2} 1 - finsupp.single {3} 1
```

and consider a basis $F = \{f_1, f_2\}$. Then by applying the function to F:
`#eval bb_criterion [f1, f2]` evaluates to `ff`. Thus F is not a Gröbner basis.

3.5 Buchberger's Algorithm

Once all necessary definitions and operations have been introduced, Buchberger's algorithm for the computation of a Gröbner basis for an ideal I can also be implemented. The ideal, is implemented as a list of multivariate polynomials and the algorithm will output a new list of polynomials, being the Gröbner basis.

```
meta def bb_alg_ax: list (my_mvpolynomial σ α) -> list (my_mvpolynomial σ α)
| [] := []
| (h::t) :=
    ((h::t) : list (my_mvpolynomial σ α)).foldr(λ p l,
    (((h::t): list (my_mvpolynomial σ α)).foldr(λ p1 l1,
    let remainder := (long_div (s_polynomial p p1) (h::t)) in
        if (remainder ≠ 0 ∧ remainder ∉ ((h::t) : list (my_mvpolynomial σ α)))
          then bb_alg_ax ((h::t) ++ [remainder])
        else l1))
    (l))
    (h::t)
```

The algorithm above considers two cases according to the input provided. An empty list (the trivial ideal) produces an empty list. On the contrary, given a nonempty list of polynomials, the algorithm computes the S-polynomial for each pair in the list. If the long division operation of the S-polynomial with the elements in the list has a nonzero remainder and not contained in the list, then this remainder is added to the current list of polynomials. The operation is repeated for the new list, until for all pairs elements included in the list, the remainder of the S-polynomial is 0.

Finally, for the final definition of Buchberger's algorithm, we consider also the condition for the computation of a Gröbner basis, that is $I \setminus \{0\}$

```
meta def bb_alg: list (my_mvpolynomial σ α) -> list (my_mvpolynomial σ α)
| [] := []
| [zero_ideal] := []
| (h::t) :=
   bb_alg_ax (h::t)
```

We are now able to compute a Gröbner basis for an ideal I. As an example we can make use of the polynomials f_1 and f_2 previously defined in Section 3.4 and consider the basis $F = \{f_1, f_2\}$. We already determined by applying Buchberger's criterion that F is not a Gröbner basis. By applying Buchberger's algorithm we get:

```
#eval bb_alg [f1, f2] : [ 1{2, 3, 3} -1{},  1{2, 2, 3} -1{3},  1{3, 3} -1{2},
    1{2, 2} -1{}]
```

We can observe that the resulting basis, contains 2 more elements than the initial basis. By constructing the 2 new polynomials, f_3, f_4:

```
def f3: my_mvpolynomial ℕ ℚ := finsupp.single {3,3} 1 - finsupp.single {2} 1
def f4: my_mvpolynomial ℕ ℚ := finsupp.single {2,2} 1 - finsupp.single {} 1
```

and applying Buchberger's Criterion to the resulted basis $F = \{f_1, f_2, f_3, f_4\}$:

`#eval bb_criterion [f1,f2,f3,f4]` evaluates to tt.

Thus for all pairs $f_i, f_j \in F$, $\overline{S(f_i, f_j)}^G = 0$ and thus F is a Gröbner basis.

4 Conclusions

In this chapter I discuss some conclusive thoughts, regarding possible improvements to this project, possible further additions and the usability of Lean.

4.1 Implementation

There are several possible improvements that could be made to this project:

Naming The names given to several operations, such as `my_mvpolynomial.l_t`, could be reconsidered for improved clarity and for uniform possible merging with the rest of Lean's standard library.

Formalization of proofs Despite the implementation, more work needs to be done to complete the multiplication property of the lex order, the formalization of the well founded relation used to prove the termination of the Long Division algorithm. It can also be proved in future time, the correctness of Buchberger's algorithm.

Commenting Further or more informational comments could be added, making clear the purpose of each definition. Although this generally leads to longer files, comments can help clarify aspects of the operations and can be beneficial especially to new users of *mathlib*.

Optimization The code's efficiency can be improved by shortening the length of the code, cleaning up redundant or unused code and possibly by reducing the number of the recursive calls for each operation.

4.2 Further implementations

Given the importance and applications of Gröbner bases but also the property that they can often be bigger than necessary, it would be constructive and interesting to formally implement and verify the definition of minimal Gröbner basis. The implementation could also be extended to include the reduced Gröbner bases, given their uniqueness for any given ideal and monomial ordering.
In addition more monomials orderings (grlex, rgrlex e.t.c) could be implemented and formalized.

4.3 Lean's usability

My only contact with a proof assistant tool was during the logic and modeling course, for which practical exercises of natural deduction had to be resolved. I have no prior experience with other proof assistant tools similar to Lean and formal verification, while my impression before using Lean was that in order for a user to handle adequately a proof assisting tool, must be a very skilled mathematician with advanced programming skills. Nevertheless, despite some initial frustration due to lack of experience with Lean's programming language and mainly because of the unfamiliarity with its standard library, once the concepts and steps for a formalization became clearer and myself more acquainted with the language and *mathlib*, the project was built fluently and in a structured way.

However, the usability of Lean could be improved. I encountered difficulties with the installation of Lean's *package manager* on windows and the available documentation alone was not detailed enough for me to overcome them. The documentation provided [11], is useful and instructive, introducing the reader to the development and verification of proofs in Lean. The book however, offers a small amount of exercises and practical examples for the user to practice. Moreover, after reading it, I got the impression that for inexperienced users, reading it by chapter order maybe is not the optimal way. It could be beneficial, for introducing new users and acquiring faster familiarization with Lean and formalization, a tutorial that maybe offers less in depth information but faster acquaintance with the language and the most important aspects of formalization, along with exercises of increasing difficulty.

Besides my inexperience with proof assistant tools, almost all the material reported in this thesis was new to me. Consequently, I am very confident that any user with some mathematical background and not necessarily a mathematician, but interested in formalizing definitions, can learn to use Lean within a reasonable time efficiently. Moreover, Lean could have a valuable use for educational purposes within the ambient of a classroom, for practicing purposes and homework exercises. Due to the benefits of formal verification tools in general and the characteristics of flexibility, availability and simplicity of Lean, it can be an important tool for students to improve the clarity of the treated arguments and the logical rules, pay attention to important details which otherwise can be skipped, acquire faster familiarity with the aspects of formalization and added flexibility for providing trustworthy proofs and increasing confidence on the subject.

For the aforementioned reasons, I would encourage the use of Lean besides within the

mathematical and logic community, within classrooms for teaching purposes and also to any amateur enthusiast with a clear goal to implement and or formalize.

References

[1] Bruno Buchberger. Ein Algorithmus zum Auffinden der Basiselemente des Restlasseringes nach einem nulldimensionalen Polynomideal. (German). *Ph.D thesis*, 1965.

[2] William W. Adams and Philippe Loustaunau. *An introduction to Gröbner bases.*, volume 3. Providence, RI: American Mathematical Society, 1994.

[3] Michael Brickenstein, Alexander Dreyer, Gert-Martin Greuel, Markus Wedler, and Oliver Wienand. New developments in the theory of gröbner bases and applications to formal verification. *Journal of Pure and Applied Algebra*, 213(8):1612 – 1635, 2009. Theoretical Effectivity and Practical Effectivity of Gröbner Bases.

[4] David A.R. Wallace. *Groups, Rings and Fields.* Springer-Verlag, Berlin, Heidelberg, 2001.

[5] David A. Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics).* Springer-Verlag, Berlin, Heidelberg, 2007.

[6] Leonardo Mendonça de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. Elaboration in dependent type theory. *CoRR*, abs/1505.04324, 2015.

[7] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29, August 2017.

[8] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. *CoRR*, abs/1701.04391, 2017.

[9] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). volume 9195, pages 378–388, 08 2015.

[10] D. Selsam, P. Liang, and D. L. Dill. Certigrad [bug-free machine learning on stochastic computation graphs]. https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

[11] J. Avigad, L. de Moura, and S. Kong. Theorem Proving in Lean. https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

[12] Lean theorem prover. https://leanprover.github.io/.