

A FUNCTIONAL COMPUTER ALGEBRA SYSTEM FOR POLYNOMIALS

BY

THOMAS MEEK

A Thesis Submitted to the Graduate Faculty of
WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES
in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Mathematics

May, 2023

Winston-Salem, North Carolina

Approved By:

W. Frank Moore, Ph.D., Advisor

Ellen Kirkman, Ph.D., Chair

William Turkett, Ph.D.

Acknowledgments

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetur id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Table of Contents

Abstract	iv
Chapter 1 Introduction	1
Chapter 2 Mathematical Background.....	4
2.1 Polynomials	5
2.1.1 Monomials	6
2.1.2 Ideals	11
2.1.3 Gröbner bases	13
2.2 Categories	15
Chapter 3 Functional Programming.....	20
3.1 Types and Kinds	25
3.2 Functors and Monads	30
Chapter 4 The Polynomial-Algorithms Package.....	31
4.1 The Polynomial Type	31
4.2 The Division Algorithm	31
4.3 Buchberger's Algorithm	32
4.4 Efficiency	32
Chapter 5 Conclusion.....	33
Bibliography	34
Curriculum Vitae	35

Abstract

The `polynomial-algorithms` package is a computer algebra system written in Haskell. This package implements a `Polynomial` type for use in several algorithms including a recursive version of Buchberger's algorithm for finding the reduced Gröbner basis of a polynomial ideal. As it is written in Haskell, the `polynomial-algorithms` package naturally uses a purely functional design philosophy. We will contrast this with imperative approaches using more traditional languages.

Chapter 1: Introduction

The theory of polynomials is a well-studied field of mathematics, rich with results which provide tools for mathematicians in various areas. Polynomial functions are among the simplest continuous functions to work with analytically or algebraically. Computationally, they are unquestionably the most important class of functions. It is then imperative to have a robust collection of software that can assist the mathematicians who study them. Many tools currently exist for this purpose, from dedicated high-level languages like Macaulay2 and Singular to libraries for general purpose languages like SageMath and SimPy. These software collections are referred to as *computer algebra systems* or CAS's and while most have capabilities beyond polynomial manipulation, it is this feature that is the focus here.

The implementation of these tools varies greatly. The Macaulay2 engine is written in C++ [10] while SageMath is built with Python [16], which in turn often calls C. Their respective interfaces often use a mixture of procedural, object-oriented, functional, and declarative design. This certainly makes sense in the case of SageMath as Python itself hybridizes these approaches. If a CAS is a package for a general purpose language like Python, it should obey the design principles of that language. For dedicated CAS's like Macaulay2 though, this design choice bears consideration. There is a tendency in modern languages to incorporate features which favor a variety of paradigms. The popularity of hybrid languages like Python and JavaScript has encouraged most newer languages like Rust and Go to follow in their footsteps. There are good reasons for this, however there are also benefits to sticking to a single paradigm.

It is true that these paradigms are more a philosophical way of approaching soft-

ware design than a well-defined feature of a language and given that, no language can be said to be strictly single-paradigm. It is clear however that certain languages lend themselves to being more compatible with one paradigm than others. For example, Java is generally seen as a paragon of object-oriented programming. Every module contains exactly one class (with potentially many subclasses). Even the drivers are classes and must be instantiated before any program can be executed. One can use Java in a functional way, but doing so defeats the purpose of using Java. This is in contrast to a language like Python where it is difficult to escape the use of objects and classes, but higher order functions like folds and maps are used frequently, as are list comprehensions and other hallmarks of functional programming. At the same time, most programmers' first experience with Python is in writing plain procedural scripts. The freedom offered by such a language enables one to choose the most appropriate approach to the current task, but in doing so, it takes away the cohesiveness and predictability of a more *pure* language like Java.

Similarly to the way Java is considered a paragon of the object-oriented school, Haskell is often one of the first two languages people think of when one mentions functional programming, with the other of course being Lisp. The functional style offers many advantages over a procedural or object-oriented approach as evidenced in how much modern languages have borrowed from Haskell. Rust's type system strongly resembles that of Haskell [14] and modern JavaScript libraries like React leverage functional programming more than any other paradigm.

Haskell and the functional programming style are particularly well-suited to mathematics. The notion of a 'function' in procedural programming is a bit of a misnomer. Mathematically, a function f is a domain X , a codomain Y , and some subset of $X \times Y$ called the graph of f . What is the domain of the `print()` function in Python? What is its codomain? What is its graph? These inconsistencies vanish in Haskell. A

Haskell function *is* a mathematical function. For this reason, mathematicians often regard the functional paradigm as the natural choice for mathematical software. What better way to determine the injectivity of a function than with a function whose injectivity can be determined?!

It is in this spirit that the **polynomial-algorithms** package was written. Taking advantage of the benefits offered by Haskell (as well as a few language extensions), this package offers a Polynomial type that is itself built on Monomial and Coefficient types. Several algorithms, most notably a modified version of Buchberger's algorithm to find a reduced Gröbner basis for a list of polynomials, are featured in this software. In addition, its scalability and modularity should provide mathematicians with an invaluable tool when analyzing polynomials.

Chapter 2: Mathematical Background

One thing that sets functional programming apart from imperative paradigms is its reliance on mathematics. Writing algorithms or any sufficiently complex code will always require some familiarity with mathematics, and the best and most efficient code is often written by programmers competent in mathematical reasoning. However, this quality is even more important when working in a language like Haskell. This fact may be seen just by reading documentation. A good technical writer will always write to their audience. If one is writing a UDP server in Java, it may be assumed that anyone maintaining that server will be familiar with terms like *port*, *buffer*, or *packet*, so they may appear frequently in documentation. If one is writing an operating system in C, terms like *fork* and *process ID* are ubiquitous enough to appear in documentation without explanation. However, one is unlikely to encounter mentions of algebraic ring theory in the documentation for such projects. When using Haskell, this is quite common [1].

It is therefore imperative (pun intended) for a prospective Haskell¹ to have a working knowledge of several areas of mathematics not usually necessary for writing programs in a procedural or object-oriented style. This is doubly important when using Haskell to write a CAS. The fact that the recommended backgrounds for the problem domain and the solution domain coincide is further evidence that Haskell is indeed a good fit for such a task.

As any software engineer will tell you, having a firm understanding of mathematical logic can be helpful in writing and debugging code. Boolean algebra is essential for the standard control flow mechanisms used in all languages. De Morgan's laws

¹The term *Haskeller* refers to a regular user of Haskell.

and other properties of first order logic can assist in cleaning up messy functions. It is well-known that a familiarity with combinatorics and graph theory can be helpful for those writing abstract data types or analyzing networks, and linear algebra and multivariate calculus are essential for working with 3D graphics. However, abstract algebra and category theory are rarely covered in the standard curriculum for a computer science degree.

When using Haskell, a working knowledge of algebraic group theory and ring theory, while not strictly necessary, can provide useful insight into why certain typeclasses and functions behave the way they do. When writing a CAS, this recommendation is upgraded to a strict requirement. Even more central in the design of Haskell and other functionally-minded languages is the presence of the ever-intimidating category theory. Even by theoretical mathematicians, this subject is often referred to as *abstract nonsense* [13]. Having a reputation as being among the most abstruse of mathematical topics, the mere fact that something as concrete and useful as writing software could employ such a concept is itself remarkable.

Using such a mathematically inclined language to implement a CAS is clearly an endeavour that requires a thorough understanding of the mathematics at play. Of particular importance is the theory of polynomials.

2.1 Polynomials

Polynomials in a single variable with real coefficients are a familiar object of study, not only to professional mathematicians, but to anyone who has taken a high school math class. They are easy to understand and work with. Finding their roots, taking their derivatives, and analyzing their graphs are among the easier tasks a mathematician will attempt. We even estimate arbitrary analytic functions as polynomials via Taylor's theorem. Many of the algorithms that form the backbone of low-level

software are based on this idea. It should come as no surprise then that generalizing these familiar objects is a popular practice. Using fields other than \mathbb{R} is a natural first step. In fact, when using an algebraically closed field like \mathbb{C} , polynomials behave even more nicely than they did over \mathbb{R} as we don't have to wonder how many roots a degree n polynomial may have; the answer is always n (up to multiplicity).

The next natural generalization is to allow for multiple (though still finitely many) variables. This generalization does introduce some complexity. For one thing, the idea of ordering the terms, which was taken for granted in the single variable case, now becomes a nontrivial discussion. The tools from multivariate calculus can be helpful in understanding the analytic nature of multivariate polynomials, while an introductory course in abstract algebra or algebraic geometry often addresses the more algebraic concerns by introducing term orders, algorithms, and the relationship between the ideals containing these objects and the affine varieties they generate.

2.1.1 Monomials

There are two competing ways to view a polynomial. The first is as a function from some ring into itself. It is this view that is usually encountered first, and for most analytic purposes, this is sufficient. The other way is as a formal linear combination of indeterminates. This is the view that we prefer in this discussion. These indeterminates comprise a free monoid², and (at least for now) we may assume the monoidal operation to be commutative. We use the term *monomial* when referring to elements of a free monoid in the context of polynomials.

In this view, the phrase *polynomial in n variables* really means a linear combination of monomials drawn from a free monoid with n distinct generators. Each generator is called a variable. When dealing with polynomials in a single variable, we

²Recall that a monoid is a set with an associative binary operation and an identity element. Being free means there are no unforced relationships between elements. For example, $x \neq y^n$ for any n .

usually use the symbol x to refer to the generator of this monoid. So the polynomial f defined as

$$f(x) = x^2 - 8x + 15$$

is a formal linear combination of the monomials x^2 , x , and 1 taken from the free monoid $\langle x \rangle$ with weights $1, -8$, and 15 . In the case of a three variable polynomial, the monoid used is generated by three elements. Here, there are two competing notational conventions. The first is to denote the generators as x , y , and z . The second is to denote the generators as x_1 , x_2 , and x_3 . The former is often more readable, but the latter more easily generalizes to n variables.

This viewpoint of polynomials as a formal combination of monomials rather than a function may be summed up in the following few definitions. Note that these and the rest of the definitions in this section are courtesy of Cox, Little and O'Shea [8]:

Definition 1. A **monomial** in x_1, \dots, x_n is a product of the form

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

where all of the exponents $\alpha_1, \dots, \alpha_n$ are nonnegative integers. The **total degree** of this monomial is the sum $\alpha_1 + \cdots + \alpha_n$.

We can simplify the notation for monomials as follows: let $\alpha = (\alpha_1, \dots, \alpha_n)$ be an n -tuple of nonnegative integers. Then we set

$$x^\alpha = x_1^{\alpha_1} \cdot x_2^{\alpha_2} \cdots x_n^{\alpha_n}.$$

When $\alpha = (0, \dots, 0)$, note that $x^\alpha = 1$. We also let $|\alpha| = \alpha_1 + \cdots + \alpha_n$ denote the total degree of the monomial x^α .

Definition 2. A **polynomial** f in x_1, \dots, x_n with coefficients in a field k is a finite linear combination (with coefficients in k) of monomials. We will write a polynomial

f in the form

$$f = \sum_{\alpha} a_{\alpha} x^{\alpha}, \quad a_{\alpha} \in k,$$

where the sum is over a finite number of n -tuples $\alpha = (\alpha_1, \dots, \alpha_n)$. The set of all polynomials in x_1, \dots, x_n with coefficients in k is denoted $k[x_1, \dots, x_n]$.

Definition 3. Let $f = \sum_{\alpha} a_{\alpha} x^{\alpha}$ be a polynomial in $k[x_1, \dots, x_n]$.

- (i) We call a_{α} the **coefficient** of the monomial x^{α} .
- (ii) If $a_{\alpha} \neq 0$, then we call $a_{\alpha} x^{\alpha}$ a **term** of f .
- (iii) The **total degree** of $f \neq 0$, denoted $\deg(f)$, is the maximum $|\alpha|$ such that the coefficient a_{α} is nonzero. The total degree of the zero polynomial is undefined.

One observation is that the set $k[x_1, \dots, x_n]$ forms a ring under the standard polynomial addition and multiplication operations. In the case where $n = 1$, the ring $k[x]$ contains the familiar polynomials in a single variable. An important question that often arises when working with polynomials in more than one variable is how to define the leading term. Given a nonzero polynomial $f \in k[x]$, let

$$f = c_0 x^m + c_1 x^{m-1} + \dots + c_m,$$

where $c_i \in k$ and $c_0 \neq 0$ (thus $\deg(f) = m$). Then we say that $c_0 x^m$ is the **leading term** of f and write $\text{LT}(f) = c_0 x^m$. The leading term of a polynomial is a surprisingly essential characteristic. For example, when executing single variable polynomial long division, the first step is to make sure the polynomial is expressed with its leading term first, and at each subsequent step, that must remain the case; but what about when there are multiple variables? Given the polynomial $g \in k[x, y, z]$ defined as

$$g = xy^2z^3 + x^5 + x^3y^2z,$$

what is the leading term of g ? We will have to be careful about how we define the leading term in this case, as there is not just one obvious way.

This leads us to a discussion of *monomial orderings*. There are uncountably many ways to order the monomials in a free monoid, but the majority of them will not be compatible with polynomial multiplication in the way we would like. The orderings that we may use must satisfy a few special properties, espoused in the following definition [8]:

Definition 4. A **monomial ordering** on $k[x_1, \dots, x_n]$ is a relation $>$ on the set of monomials x^α , $\alpha \in \mathbb{Z}_{\geq 0}^n$ satisfying:

- (i) $>$ is a total ordering³.
- (ii) If $x^\alpha > x^\beta$ and $\gamma \in \mathbb{Z}_{\geq 0}^n$, then $x^\alpha x^\gamma > x^\beta x^\gamma$.
- (iii) $>$ is a well-ordering.

It turns out that the third condition above is equivalent to two other statements that are easier to work with.

Theorem 5. *Let X be a commutative free monoid and suppose the first two conditions in the definition above are satisfied. Then the following are equivalent:*

1. $>$ is a well-ordering on X .
2. Every strictly decreasing sequence in X eventually terminates.
3. $x^\alpha \geq 1$ for all $\alpha \in \mathbb{Z}_{\geq 0}^n$.

For a proof of this theorem, see Cox, Little and O'Shea [8]. This allows us to show that certain algorithms terminate by showing that some term strictly decreases at each step of the algorithm.

³Recall $>$ is a total ordering if for all α, β exactly one of $x^\alpha > x^\beta$, $x^\alpha = x^\beta$, or $x^\beta > x^\alpha$ is true.

Now that we have a well-defined way to order monomials, we are ready to define a few more terms.

Definition 6. Let $f = \sum_{\alpha} a_{\alpha} x^{\alpha}$ be a nonzero polynomial in $k[x_1, \dots, x_n]$ and let $>$ be a monomial order.

(i) The **multidegree** of f is

$$\text{multideg}(f) = \max(\alpha \in \mathbb{Z}_{\geq 0}^n \mid a_{\alpha} \neq 0)$$

(the maximum is taken with respect to $>$).

(ii) The **leading coefficient** of f is

$$\text{LC}(f) = a_{\text{multideg}(f)} \in k.$$

(iii) The **leading monomial** of f is

$$\text{LM}(f) = x^{\text{multideg}(f)}.$$

(iv) The **leading term** of f is

$$\text{LT}(f) = \text{LC}(f) \cdot \text{LM}(f).$$

There are many monomial orderings that are used frequently in research, however we will limit our attention to three specific examples. The first such example is perhaps the most intuitive.

Definition 7 (Lexicographic Order). Let $\alpha = (\alpha_1, \dots, \alpha_n)$ and $\beta = (\beta_1, \dots, \beta_n)$ be in $\mathbb{Z}_{\geq 0}^n$. We say $x^{\alpha} >_{\text{Lex}} x^{\beta}$ if the leftmost nonzero entry of the vector difference $\alpha - \beta \in \mathbb{Z}^n$ is positive.

The Lexicographic order is the order used in most dictionaries. We begin with an ordering of the variables. The greater monomial is the one with the larger exponent in the first variable. If those exponents are the same, we instead calculate the Lexicographic order without the highest variable. In our previous example, $g = xy^2z^3 + x^5 + x^3y^2z$, we see that the lead term under the Lexicographic order where $x > y > z$ is $\text{LT}(g) = x^5$. By convention, when denoting variables with subscripts, we take $x_1 > x_2 > x_3 > \dots$.

Our next monomial ordering is built out of the Lexicographic order.

Definition 8 (Graded Lex Order). Let $\alpha, \beta \in \mathbb{Z}_{\geq 0}^n$. We say $x^\alpha >_{GLex} x^\beta$ if $|\alpha| > |\beta|$ or $|\alpha| = |\beta|$ and $x^\alpha >_{Lex} x^\beta$.

This means that the Graded Lex order first orders by total degree, but breaks ties with the Lexicographic order. In our running example using the Graded Lex order, we see that $\text{LT}(g) = x^3y^2z$.

Our third and final monomial ordering is less intuitive than the previous two.

Definition 9 (Graded Reverse Lex Order). Let $\alpha, \beta \in \mathbb{Z}_{\geq 0}^n$. We say $x^\alpha >_{GRevLex} x^\beta$ if $|\alpha| > |\beta|$ or $|\alpha| = |\beta|$ and the rightmost nonzero entry of $\alpha - \beta \in \mathbb{Z}^n$ is negative.

Here, we are still ordering first by total degree, but we break ties in a manner that is in a way the *double reversal* of Lexicographic order. Using the Graded Reverse Lex order, the lead term of g is xy^2z^3 . As unintuitive as the Graded Reverse Lex order may seem, it turns out that this ordering is often the most efficient in many algorithms, including some we will explore here.

2.1.2 Ideals

In general, when R is a commutative ring, a subring I (not necessarily with identity) is an ideal if $ra \in I$ for all $r \in R$ and $a \in I$. For the polynomial ring $k[x_1, \dots, x_n]$

this means that an ideal is a nonempty subset closed under subtraction which absorbs polynomials by multiplication. For example, let $S \subseteq k[x]$ be the set of polynomials with no constant or linear terms, so that $0 \in S$ and for any nonzero $f \in S$, the lowest degree term of f has degree at least two. Then S is easily seen to be an ideal of $k[x]$. In fact, S is what is known as a *principal ideal* because S is *generated* by a single polynomial, namely the polynomial x^2 . This means that any polynomial in S may be formed by multiplying x^2 by some polynomial in $k[x]$. More generally, an ideal $I \subseteq k[x_1, \dots, x_n]$ is said to be generated by a set B of polynomials if every $f \in I$ may be expressed as $f = b_1 f_1 + \dots + b_t f_t$ for some $b_1, \dots, b_t \in B$ and some $f_1, \dots, f_t \in k[x_1, \dots, x_n]$. In this case, we write $I = \langle B \rangle$. In our example above, we would write $S = \langle x^2 \rangle$. It turns out that *every* ideal in $k[x]$ is generated by a single polynomial, making this ring quite important in algebra and number theory. When dealing with multiple variables, we have no such luck. There is no single polynomial that generates the ideal $\langle x, y \rangle \subseteq k[x, y, z]$. However, it is true that every polynomial ideal is generated by *some* set since an ideal will generate itself. What is more impressive and not quite as obvious is the fact that every polynomial ideal is generated by some *finite* set. This result was first proved by David Hilbert in 1890 and is known as *Hilbert's Basis Theorem* [11].

Combining our notion of the leading term of a polynomial with ideals in a polynomial ring leads us to the following definition.

Definition 10. Let $I \subseteq k[x_1, \dots, x_n]$ be an ideal other than $\{0\}$, and fix a monomial ordering on $k[x_1, \dots, x_n]$. Then:

- (i) We denote by $\text{LT}(I)$ the set of leading terms of nonzero elements of I . Thus,

$$\text{LT}(I) = \{cx^\alpha \mid \text{there exists } f \in I \setminus \{0\} \text{ with } \text{LT}(f) = cx^\alpha\}.$$

- (ii) We denote by $\langle \text{LT}(I) \rangle$ the ideal generated by the elements of $\text{LT}(I)$.

If $I \subseteq k[x_1, \dots, x_n]$ is an ideal then by Hilbert's Basis Theorem, there is some finite set of polynomials $\{b_1, \dots, b_t\} \subseteq k[x_1, \dots, x_n]$ such that $I = \langle b_1, \dots, b_t \rangle$. Since $b_1, \dots, b_t \in I$, it is clear that $\langle \text{LT}(b_1), \dots, \text{LT}(b_t) \rangle \subseteq \langle \text{LT}(I) \rangle$. However, this containment may be proper. Only in a very special case do we achieve equality of these two sets.

2.1.3 Gröbner bases

We mentioned polynomial long division before when making the case for why a monomial ordering is necessary. Another challenge when working with multiple variables is that, even when we have the requisite ordering to perform long division, the quotient and remainder may depend on the order of the divisors. When dividing f by the ordered list (g_1, g_2, g_3) , we may get a different result from when dividing f by the ordered list (g_2, g_3, g_1) . While the quotient is unfortunately doomed to this fate, there is a way we may guarantee the uniqueness of at least the remainder. It turns out this uniqueness is crucial for many of the theorems and algorithms that mathematicians rely on.

Definition 11. Fix a monomial order on the polynomial ring $k[x_1, \dots, x_n]$. A finite subset $G = \{g_1, \dots, g_t\}$ of an ideal $I \subseteq k[x_1, \dots, x_n]$ different from $\{0\}$ is said to be a **Gröbner basis** if

$$\langle \text{LT}(g_1), \dots, \text{LT}(g_t) \rangle = \langle \text{LT}(I) \rangle.$$

Using the convention that $\langle \emptyset \rangle = 0$, we define the empty set \emptyset to be the Gröbner basis of the zero ideal $\{0\}$.

Another way to state this definition is that the set $G = \{g_1, \dots, g_t\}$ is a Gröbner basis for I if the leading term of any element of I is divisible by one of the $\text{LT}(g_i)$. As one would expect from the terminology, a Gröbner basis is a generating set for the ideal

I in the definition above and it can be shown that every ideal has a Gröbner basis. This is lucky because much of modern polynomial theory and algebraic geometry depends on the existence of such a basis. This is partly because the Gröbner basis is the set we needed to guarantee uniqueness of the remainder in the division algorithm.

Another feature of Gröbner bases is a solution to the so called *ideal membership problem*. For a proof of the following theorem, see [8].

Theorem 12. *Let $G = \{g_1, \dots, g_t\}$ be a Gröbner basis for an ideal $I \subseteq k[x_1, \dots, x_n]$ and let $f \in k[x_1, \dots, x_n]$. Then $f \in I$ if and only if the remainder on division of f by G is zero.*

The remainder is sometimes called the **normal form** of f . We will use the following notation for the remainder.

Definition 13. We will write \overline{f}^F for the remainder on division of f by the ordered s -tuple $F = (f_1, \dots, f_s)$. If F is a Gröbner basis for $\langle f_1, \dots, f_s \rangle$, then we can regard F as a set (without any particular order).

One application of the division algorithm is Buchberger's algorithm for finding a Gröbner basis for the ideal $\langle f_1, \dots, f_t \rangle$. The idea behind this algorithm is to start with a list (f_1, \dots, f_t) , determine all polynomials in $\langle f_1, \dots, f_t \rangle$ that may have a lead term that isn't divisible by any of the $\text{LT}(f_i)$, and add to the list the ones that aren't. To find a polynomial with lead term that isn't divisible by any of the $\text{LT}(f_i)$, we need to construct a polynomial in our ideal that has a potentially different lead term from the generators. We do this by calculating *S-polynomials*. The way we check if our set can generate this *S-polynomial* is via the division algorithm.

Definition 14. Let $f, g \in k[x_1, \dots, x_n]$ be nonzero polynomials.

- If $\text{multideg}(f) = \alpha$ and $\text{multideg}(g) = \beta$, then let $\gamma = (\gamma_1, \dots, \gamma_n)$, where

$\gamma_i = \max(\alpha_i, \beta_i)$ for each i . We call x^γ the **least common multiple** of $\text{LM}(f)$ and $\text{LM}(g)$, written $x^\gamma = \text{lcm}(\text{LM}(f), \text{LM}(g))$.

- The **S-polynomial** of f and g is the combination

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

(Note that we are inverting the leading coefficients here as well.)

This definition leads to a crucial result in the theory of Gröbner bases.

Theorem 15 (Buchberger’s Criterion). *Let I be a polynomial ideal. Then a basis $G = \{g_1, \dots, g_t\}$ of I is a Gröbner basis of I if and only if for all pairs $i \neq j$, the remainder on division of $S(g_i, g_j)$ by G (listed in some order) is zero.*

Gröbner bases for ideals in polynomial rings were introduced by Bruno Buchberger in his PhD thesis [5] and named by him in honor of Wolfgang Gröbner, Buchberger’s thesis adviser. This construction sharply influenced the direction of computational algebra for the next half century. It is then no surprise that finding a Gröbner basis for a polynomial ideal in the most efficient way is a task of central importance in the field. Bruno Buchberger also developed the eponymous Buchberger’s algorithm and since then, many tweaks and improvements have been made, but the basic logic of the algorithm remains largely unchanged.

2.2 Categories

Over the last seventy-five years, the landscape of mathematics research has shifted in many ways. One of these ways is the ubiquity of categories in many mathematical fields. Modern mathematicians dealing with sheaves or tensors are more likely to use their categorical definitions than classical constructions. There are increasingly many

attempts to introduce category theory to undergraduate mathematics students [3]. Still, category theory has not shed its reputation for being overly abstract and useless outside of the purest of mathematical endeavors. To be fair, the first criticism is justified. Category theory is indeed abstract. Perhaps this is why the aforementioned attempts to introduce this theory to a less experienced audience have not yet become standard practice. The second criticism however is simply unfounded.

Apart from providing more elegant solutions to mathematical problems found in other fields, and providing a unified framework with which to explore the relationships between otherwise unrelated disciplines, category theory does have actual applications outside of pure mathematics. For an example, one need not look any further than the Haskell language. One of the most commonly used typeclasses is the Functor typeclass, which specifies a mechanism for *mapping over* some structure. The idea of a functor is taken from category theory and is essential for effectively writing software in the Haskell language. The remaining definitions in this chapter are largely taken from Emily Riehl’s *Category Theory in Context* [15].

Definition 16. A **category** \mathbf{C} is a collection of **objects** and **morphisms** such that:

- Each morphism has specified **domain** and **codomain** objects. We denote the collection of morphisms with domain a and codomain b as $\text{Hom}(a, b)$. If this collection is a set⁴, we call $\text{Hom}(a, b)$ the **hom set** of (a, b) .
- Each object x has a designated **identity morphism** $1_x \in \text{Hom}(x, x)$.
- If $f \in \text{Hom}(a, b)$ and $g \in \text{Hom}(b, c)$, then there is a morphism $gf \in \text{Hom}(a, c)$.

In this case, we say f and g are **composable** and gf is their **composite**.

This data is subject to the following two axioms:

⁴The axioms of set theory may prevent this.

- For any $f \in \text{Hom}(a, b)$, the composites $1_b f$ and $f 1_a$ are both equal to f .
- For any composable triple of morphisms f, g, h , the composites $h(gf)$ and $(hg)f$ are equal.

The prototypical example of a category is **Set** whose objects are sets and whose morphisms are the functions between those sets. In many categories, the morphisms are set-functions of some kind, but that is not always the case.

Example 17. Recall that a poset is a set along with a relation that is reflexive, transitive, and antisymmetric. Let (S, \leq) be a poset. Then we may define a category **S** that has the elements of S as its objects. If $a \leq b$ then let $\text{Hom}(a, b)$ be the singleton consisting of the element $(a, b) \in S \times S$. Otherwise, let $\text{Hom}(a, b) = \emptyset$. Define the composition law as $(a, b)(b, c) = (a, c)$. As each hom set has at most a single element, there is only one choice for the identity morphism.

If (a, b) and (b, c) both exist, then we must have that $a \leq b \leq c$, so since \leq is transitive, (a, c) exists and so composition is well defined. Since \leq is reflexive, $(a, a) \in \text{Hom}(a, a)$ and

$$(a, a)(a, b) = (a, b) = (a, b)(b, b),$$

so the (a, a) does serve as the identity on a . Also observe that

$$((a, b)(b, c))(c, d) = (a, c)(c, d) = (a, d) = (a, b)(b, d) = (a, b)((b, c)(c, d)),$$

so composition is associative. This shows that **S** satisfies the axioms for a category.

It is common in mathematics to introduce some kind of object that is only important because of the functions between objects of that kind. Linear algebra may be the study of vector spaces, but the important part is the linear maps. Topology may be concerned with open sets, but it is the continuous maps that topologists really

care about. Group homomorphisms are in a sense are much more important than groups themselves. It is no different with categories. The interesting part is the maps between categories. The even more interesting part is the maps between those maps.

Definition 18. A **functor** $\mathcal{F} : \mathcal{C} \rightarrow \mathcal{D}$ between categories \mathcal{C} and \mathcal{D} consists of the following data:

- An object $\mathcal{F}x \in \mathcal{D}$ for each object $x \in \mathcal{C}$.
- A morphism $\mathcal{F}f \in \text{Hom}(\mathcal{F}a, \mathcal{F}b)$ for each morphism $f \in \text{Hom}(a, b)$.

These assignments are required to satisfy the following axioms:

- For any composable pair of morphisms $f, g \in \mathcal{C}$, $(\mathcal{F}f)(\mathcal{F}g) = \mathcal{F}(gf)$.
- For each object $x \in \mathcal{C}$, $\mathcal{F}(1_x) = 1_{\mathcal{F}x}$.

We call a functor $\mathcal{F} : \mathcal{C} \rightarrow \mathcal{C}$ between a category and itself an **endofunctor**. The category of endofunctors on \mathcal{C} is denoted $\mathcal{C}^{\mathcal{C}}$. The morphisms in the category $\mathcal{C}^{\mathcal{C}}$ are examples of *natural transformations*.

Definition 19. Given functors $\mathcal{F}, \mathcal{G} : \mathcal{C} \rightarrow \mathcal{D}$, a **natural transformation** $\tau : \mathcal{F} \Rightarrow \mathcal{G}$ is a map from objects in \mathcal{C} to morphisms in \mathcal{D} such that $\tau_x \in \text{Hom}(\mathcal{F}x, \mathcal{G}x)$ making the following diagram commute for all $a, b \in \mathcal{C}$ and all $f \in \text{Hom}(a, b)$.

$$\begin{array}{ccc} \mathcal{F}a & \xrightarrow{\tau_a} & \mathcal{G}a \\ \mathcal{F}f \downarrow & & \downarrow \mathcal{G}f \\ \mathcal{F}b & \xrightarrow{\tau_b} & \mathcal{G}b \end{array}$$

The idea of commuting diagrams is central to the study of category theory. It is a concise way of saying that any allowed composition of maps in the diagram will be equivalent if they share a domain and a codomain. A common categorical technique to

prove theorems is called *diagram chasing*. Diagram chasing involves finding equivalent map compositions.

We won't spend much time discussing the finer points of category theory. If teaching the fundamentals to a class of advanced undergraduate mathematics students over the course of a semester in their senior year is a bold undertaking, then explaining it in a single section of this document is a Kobayashi Maru⁵. We do however need to address one more categorical construction before proceeding.

Definition 20. A **monad** on a category \mathbf{C} consists of

- an endofunctor $\mathcal{T} : \mathbf{C} \rightarrow \mathbf{C}$,
- a **unit** natural transformation $\eta : 1_{\mathbf{C}} \Rightarrow \mathcal{T}$, and
- a **multiplication** natural transformation $\mu : \mathcal{T}^2 \Rightarrow \mathcal{T}$,

so that the following diagrams commute in $\mathbf{C}^{\mathbf{C}}$:

$$\begin{array}{ccc}
 \mathcal{T}^3 & \xrightarrow{\mathcal{T}\mu} & \mathcal{T}^2 \\
 \mu\mathcal{T} \downarrow & & \downarrow \mu \\
 \mathcal{T}^2 & \xrightarrow{\mu} & \mathcal{T}
 \end{array}
 \qquad
 \begin{array}{ccccc}
 \mathcal{T} & \xrightarrow{\eta\mathcal{T}} & \mathcal{T}^2 & \xleftarrow{\mathcal{T}\eta} & \mathcal{T} \\
 \searrow 1_{\mathcal{T}} & & \downarrow \mu & & \swarrow 1_{\mathcal{T}} \\
 & & \mathcal{T} & &
 \end{array}$$

The reason this definition is necessary will become apparent in Chapter 3.

⁵See *Star Trek II: The Wrath of Khan* (1982).

Chapter 3: Functional Programming

Among the key features of a language like Haskell is the guarantee of functional purity. A function in Haskell does exactly one thing. It returns a value. That's it. It cannot alter the state of the program. It cannot react to external state. It cannot cause anything to happen other than returning that value. This makes a Haskell program far more predictable than any program written outside of a purely functional paradigm. This is particularly useful when writing multithreaded code. The inability for a thread to alter any resource that another thread is using simplifies the task tremendously.

In this way, software written in Haskell, or indeed any purely functional language, resembles mathematics quite closely. As we have previously mentioned, a mathematical function is merely a domain set, a codomain set, and a graph. When thinking of a function as *doing something*, the only thing it *does* is transform a domain element into its associated codomain element. However, the very notion of a function doing anything is a bit of an abuse of language. Given sets X and Y , the existence of a function $f : X \rightarrow Y$ gives us a way to refer to certain elements of Y as $f(x)$ for some $x \in X$. It doesn't really preform an action. This is in contrast to a function in the computer programming sense. A function `foo` in the Python language certainly does something. When `foo` is called, it preforms its task and then it returns if and when it finishes. This is really the most fundamental difference in functional programming and imperative programming.

Recall the difference between a declarative sentence and an imperative sentence. The sentence "Steve has twelve eggs." is a declarative sentence while the sentence "Give Steve twelve eggs." is an imperative sentence. The foundations of mathemat-

ics are built on the use of statements. A statement is a declarative sentence with a well-defined (or unambiguous) truth value [4]. Theorems are statements. Definitions are statements. A proof is a collection of statements. Mathematicians deal predominantly with statements. If we encounter a question, we first rephrase it as a statement before we attempt to verify its veracity. However, imperative programmers are more accustomed to dealing with imperative sentences (indeed, it is easy to forget that this is where the term *imperative programming* comes from.) An instruction is an imperative sentence. A traditional algorithm is a sequence of imperative sentences. `print("Hello world")` is an imperative sentence. It could be said that imperative programming is as much founded on the unambiguous imperative sentence as mathematics is founded on the unambiguous declarative sentence.

The procedural and object oriented design philosophies are both considered imperative styles. Functional programming is often considered a declarative style. What this means is that a line of Java or C code is usually meant to be read as an imperative sentence, while a line of Haskell code is meant to be read as a declarative sentence. This allows Haskell syntax to resemble the syntax of mathematics much more closely than other languages. For example, say we want to construct a list containing the first ten perfect squares. In C, this task amounts to a series of instructions:

```
int squares[10];
for (int i = 0; i < 10; i++)
    squares[i] = i*i;
```

In Haskell on the other hand, instead of specifying instructions, we merely declare the existence of such a list, much like we would in mathematics:

```
squares = [x^2 | x <- [0..9]]
```

In this way, the discussion of functions having side effects becomes moot as functions don't *do* anything. They are merely maps identifying domain elements with codomain elements, just as they are in mathematics.

Because of this, the explicit temporal constraints usually present in other languages simply aren't there in Haskell. In the C version of the `squares` function above, the square of 2 is clearly evaluated before the square of 3. Whereas in the Haskell version, that may or may not be the case. Expressions are only evaluated when they are needed and not before. This is known as *lazy evaluation*. This leads to some interesting features in Haskell. One is infinite lists. When mathematicians see the expression, “Consider the set \mathbb{Z} ”, they don't run through every integer in their head and allocate infinitely many brain cells to holding the values for each integer. They just know that \mathbb{Z} is a set and know its properties. Then, once a specific integer is mentioned, they can allocate the necessary brain power to think of that specific one they need. This is how lazy evaluation works. In the following line of code, `[3..]` is the infinite list consisting of every integer greater than or equal to 3.

```
take 5 [3..]
```

The `take n` function takes the first n elements from the list it is given. In this case, we get the list `[3,4,5,6,7]`. There was no need to evaluate the 6th entry of the list, so Haskell never did.

When thinking about functions in a declarative manner, it becomes apparent that the only real difference between functions and non-functions is that a function needs an additional element to be fully evaluated. If `fn` is a Haskell function that maps something of type X to something of type Y and `x` is a term of type X , then `fn` has type $X \rightarrow Y$ and `fn x` has type Y . In Haskell, we write

```
fn :: X -> Y
fn x = y
```

This syntax should be quite familiar to mathematicians. Indeed, it is the traditional syntax of mathematics that inspired the design of Haskell syntax.

Hidden in this syntax is one of Haskell's principal features: Currying. Say we want a function that takes two integers as input and returns their sum. In C, this

would look something like the following.

```
int addTwo(int x, int y) {  
    return x + y;  
}
```

In Haskell however, there is no such thing as a function that takes two inputs. Just like mathematics, a function has a single domain, so every function only has one input variable. Now of course, we may define a function out of a product set like $f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$. In Haskell this is accomplished with tuples.

```
addTwo :: (Int,Int) -> Int  
addTwo (x,y) = x + y
```

However, there is another, more natural way to accomplish this in mathematics. Instead of using products, simply define a function from \mathbb{Z} into the set of functions $\mathbb{Z} \rightarrow \mathbb{Z}$. This is the more common way to implement such a function in Haskell.

```
addTwo :: Int -> Int -> Int  
addTwo x y = x + y
```

The strength of this approach is the ability to partially evaluate functions. If we want a function $\mathbb{Z} \rightarrow \mathbb{Z}$ that returns the sum of a number and five, we write `addTwo 5`. So `addTwo` is a function that takes an integer and returns a function and `addTwo 5` is a function that takes an integer and returns an integer. We say that `addTwo 5` is a *partially evaluated* function. We say that an expression like `addTwo 5 8` is *fully evaluated* because it is not expecting any more parameters.

Since a function can be the output of another function, it should come as no surprise that a function can also be the input to another function. Because of this, functions in Haskell are sometimes called *first class citizens*. Some functions specifically call for unevaluated functions as input. These are known as *higher order functions*. Higher order functions like `map` and `fold` are ubiquitous in languages like Haskell and are one of the nicest things about such languages. Since functions and regular values are treated much the same way, we use the word *term* to mean any expression that

may be used as the input or output of a function.

Functional programming is not without its challenges. Since every expression is a declarative statement, the concept of iterative control must be handled differently. In C, if we wish to perform a task until a certain condition is met, it is common to use a loop to achieve this.

```
int collatz(int n) {
    while (n != 1) {
        if (n % 2 == 0) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    return n;
}
```

However, “While some condition holds, perform some action” is an imperative sentence, not a declarative one, so it is not allowed in Haskell. How then do we handle iteration? The answer is with recursion¹.

```
collatz :: Int -> Int
collatz n
    | n == 1      = 1
    | isEven n    = collatz (n `div` 2)
    | otherwise   = collatz (3 * n + 1)
```

In many cases, a recursive solution is cleaner and more readable than a loop, but even the most ardent Haskeller will admit that at times, this *feature* is a limitation. One challenge that arises is translating existing algorithms, which often use loops and other imperative mechanisms, into functional ones. For an example of this, see Chapter 4 of this document.

The Haskell language is named in honor of mathematician Haskell Curry. Curry played a pivotal role in the foundations of modern computational theory and mathematical logic. The idea of function currying is named after him. It is a bit ironic then, that the mathematical foundations of the Haskell language are rooted in a paper

¹*Recursion*. Definition: See recursion.

published in 1936 that stifled arguably the most ambitious goal of Haskell Curry's PhD advisor David Hilbert [2].

There are two competing models for computation commonly used today. One is the Turing machine, first described by Alan Turing [17]. The other is λ -calculus, first described by Turing's PhD advisor, Alonzo Church [6]. Within the publications that contained the initial descriptions of these models, Church and Turing proved using their respective machinery that Hilbert's *entscheidungsproblem* has no solution. According to the Church-Turing thesis, these models are mathematically equivalent [12]. The fundamental difference between these two models is that Turing machines carry state and perform sequential actions, while λ -calculus consists of expressions that may be composed and evaluated, but carry no state. In short, Turing machines form the basis for imperative programming and λ -calculus forms the basis for functional programming. However, λ -calculus does not offer a full description of the Haskell language. The most important feature of Haskell not present in the (standard) λ -calculus is Haskell's type system.

3.1 Types and Kinds

A type system is a way of classifying terms. The term `5` has type `Int`. The term `"Hello"` has type `String`. The term `squares` from above has type `[Int]` (a list of `Int`s). The term `addTwo` from above has type `Int -> Int -> Int`. The way one explicitly annotates the type of a term is with the `::` symbol.

```
squares :: [Int]
```

It is common practice to annotate top-level functions for the sake of readability, but Haskell can infer most types without explicit annotation.

The types listed above, `Int`, `[Int]`, `String`, etc., are built in to the Haskell language. It is also possible for a user to define their own types. The way one does this

is via the `Data` keyword.

```
Data Color = Red | Blue | Green
```

In the example above, we have defined a type `Color` and three terms, `Red`, `Blue`, and `Green`, each of which is of type `Color`. If we wish to create a term of the type `Color`, we therefore have three different way of doing this. `Red`, `Blue`, and `Green` are known as *value constructors* because they construct values of type `Color`. One can also define types which depend on other types. These are called *parametric types*.

```
Maybe a = Nothing | Just a
```

The above example is slightly more complicated than the first one. On the left of the `=` sign, the variable `a` is called a *type variable* and `Maybe a` is the type we are defining. This is a polymorphic type, so `a` may be any of the types we have seen before. `Maybe Int` is a type, `Maybe Color` is a type, etc. `Maybe` by itself is known as a *type constructor*. On the right of the `=` sign, we have two value constructors, `Nothing` and `Just`. The `Just` constructor takes a parameter `a`, while the `Nothing` constructor does not. Value constructors are terms, just like functions, so we may think of `Just a` as being a fully evaluated function of type `Maybe a`. In this case, the `a` on the right must be a term whose type is the `a` on the left. For example, `Just 5` is a term of type `Maybe Int`. `Nothing` could also have type `Maybe Int`, but it may have type `Maybe String` so we need more context to be sure. `Maybe` is built in to the Haskell language, so we needn't define it ourselves. In fact, `Maybe a` is quite an important type, as we will soon see.

Type systems in programming languages are useful for ensuring that software makes sense. Suppose a function `mult` multiplies two numbers and returns their product, and say a user passes in a string and a boolean. One of three things can happen. Either the program will crash without explanation, the program will return a nonsensical answer, or the program will output a useful error message saying you can't

pass strings or booleans to `mult`. Clearly, the third option is the desired behavior. The only way that can happen is with a sufficiently robust type system.

One of the earliest examples of a type system was when Alonzo Church modified his λ -calculus to include a rudimentary type system [7]. This became known as the *simply typed λ -calculus* which is the simplest example of a typed λ -calculus. There have since been many extensions to the simply typed λ -calculus. One such extension, known as System FC, is the type system upon which Haskell is based [9]. An analysis of the features of System FC and why it is used instead of Hindley-Milner or some other type system is beyond the scope of this discussion. Suffice it to say, it is quite a powerful type system and is often cited as one of Haskell's strongest features. However, even with such a robust type system, some tasks are still impossible by default. For these scenarios, we may turn to language extensions. Language extensions are a way to extend the capabilities of Haskell, its type system, and its compiler. Sometimes these are for mere convenience or readability; enabling certain *syntactic sugars* that help to keep code clean and maintainable. Other times, these extensions can do much more. One such extension is `DataKinds`.

We have mentioned that types are a way of classifying terms, but is there a way to classify types? A *kind* is just that. A kind may be thought of as the type of a type. Most of the types we have encountered so far have the `*` kind. `Int`, `String`, `[Int]`, and `Int -> String` are all examples of this. The `*` kind consists of so called *concrete types*. That is, the types of terms. If we are to pass an expression to a function, it must be of some type whose kind is `*`. There is one type we have mentioned that is *not* of the `*` kind though. The `Maybe` type constructor has kind `* -> *`. So types only have kind `*` if they are fully evaluated. With the `DataKinds` extension, users are able to engage in what is known as *type-level* programming. This means manipulating types as though they were terms. Instead of writing functions that map terms of a

given type to terms of another type, we may write functions that map types of a given kind to types of another kind. One of the features of `DataKinds` is that whenever you define a new type, you also get a new kind. Consider our `Color` example from before.

```
Data Color = Red | Blue | Green
```

This `Data` declaration defines a new type, `Color` of kind `*`, as well as three new value constructors of type `Color`. With the `DataKinds` extension enabled, this same line of code also defines a new kind, also called `Color`, and three new types, `Red`, `Blue`, and `Green`, each of kind `Color`. This is how we define custom kinds.

Another language extension worth noting is the `GADTs` or Generalized Algebraic Datatypes extension. When used in conjunction with `DataTypes`, this allows (among other things) the use of a more explicit syntax for defining types.

```
Maybe :: * -> * where  
  Nothing :: Maybe a  
  Just :: a -> Maybe a
```

As we can see, this syntax makes explicit our notion that value constructors are functions. It also makes it easy to see that type constructors are type-level functions. With `GADTs` syntax, we annotate the value constructors with their type, and we annotate the type constructors with their kind. In the case of `Maybe`, it is probably cleaner to use the standard syntax for defining types, but in more complicated examples with more esoteric kinds, the `GADTs` syntax can make code much more readable.

Another feature of the `DataKinds` extension is type-level natural numbers. Using the `TypeLits` module, we gain access to a new kind called `Nat`. Types of the `Nat` kind are type-level natural numbers. These types can be quite useful when creating terms whose type is indexed numerically.

```
Vector :: Nat -> * where  
  MakeVector :: [Int] -> Vector n
```

In the above definition, `Vector` is a type constructor, `MakeVector` is a value constructor, and `n` is a type of the `Nat` kind which should represent the length of the vector.

If we were to define a function `VecAdd :: Vector n -> Vector n -> Vector n` that adds two vectors, we wouldn't want a user to be able to add two vectors of different length. The benefit of using `Nat` in this construction is that this is already guaranteed by the type system as `Vector 3` and `Vector 5` are different types.

One more feature of Haskell's type system worth mentioning is *typeclasses*. Typeclasses are like templates for a type, specifying some functions that terms of that class are compatible with. If we wanted to make `Color` implement the `Show` typeclass, we could implement it like this:

```
instance Show Color where
  show Red    = "Red"
  show Blue   = "Blue"
  show Green  = "Green"
```

Here, the `show` function takes a term of type `Color` and outputs a string. We could also let Haskell implement `Show` for us by altering our `Data` declaration.

```
Data Color = Red | Blue | Green deriving Show
```

In this case, Haskell derives `Show` just like we did manually above. Note that a type may only be an instance of a given typeclass if it is of the appropriate kind. The `Show` typeclass is meant for types of kind `*` so `Color`, `Int`, and `Maybe String` may be instances of `Show`, but `Maybe` by itself cannot be. When writing a function, we may specify that one of the parameters must be an instance of some typeclass.

```
showMany :: Show a => Int -> a -> [String]
showMany n a = replicate n (show a)
```

The above function takes an `Int` and some other term. Its output is a list of string representations of that second term. So `showMany 3 False` would return `["False", "False", "False"]`. The `Show => a` expression is called a *constraint*. It means that the second parameter must be an instance of the `Show` typeclass. Without that, we couldn't guarantee that `show a` made sense.

Haskell is a statically-typed language. This means that type safety is checked

at compile time, not run time. Any type-level code is evaluated at compile time, as opposed to term-level code which is evaluated at run time. This not only adds extra clarity by removing manual compatibility checks, it also increases efficiency by offloading those checks to the compiler. Even better, this can turn run time errors into compile time errors. From a debugging standpoint, this is a huge advantage.

3.2 Functors and Monads

We said before that only types of kind `*` may be instances of the `Show` typeclass. Some typeclasses are meant for other kinds though. One such typeclass is `Functor`. To be an instance of `Functor`, a type must have the `* -> *` kind. This means that `Maybe` can be a `Functor` but `Maybe Int` cannot. The `Show` typeclass specified that the `show` function can work for a given type. The `Functor` typeclass specifies the `fmap` function.

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just a) = Just (f a)
```

The `fmap` function has type `Functor f => (a -> b) -> f a -> f b`.

Chapter 4: The Polynomial-Algorithms Package

4.1 The Polynomial Type

4.2 The Division Algorithm

A few notational conventions that we will use for this section:

- Given a set T , denote¹ the set of finite ordered lists of elements of T as $[T]$. Let $\emptyset \in [T]$ denote the empty list.
- If the list X is in $[T]$, then denote the n th entry in X as x_n and let $X \setminus x_n$ refer to the list obtained by removing x_n from X and maintaining the relative order of the rest of X .
- Suppose n is given and let $\mathbb{F}[\mathbf{x}] = \mathbb{F}[x_1, \dots, x_n]$.

Let $\varphi : [\mathbb{F}[\mathbf{x}]] \times (\mathbb{F}[\mathbf{x}] \times \mathbb{F}[\mathbf{x}]) \rightarrow \mathbb{F}[\mathbf{x}] \times \mathbb{F}[\mathbf{x}]$ be the function defined as

$$\varphi(G, (p, r)) = \begin{cases} (p - \text{LT}(p), r + \text{LT}(p)) & \text{if } G = \emptyset, \\ (p - \frac{\text{LT}(p)}{\text{LT}(g_1)} g_1, r) & \text{if } G \neq \emptyset \text{ and } \text{LT}(g_1) \mid \text{LT}(p), \\ \varphi(G \setminus g_1, (p, r)) & \text{if } G \neq \emptyset \text{ and } \text{LT}(g_1) \nmid \text{LT}(p). \end{cases}$$

Let $\psi : [\mathbb{F}[\mathbf{x}]] \times (\mathbb{F}[\mathbf{x}] \times \mathbb{F}[\mathbf{x}]) \rightarrow \mathbb{F}[\mathbf{x}]$ be the function defined as

$$\psi(G, (p, r)) = \begin{cases} r & \text{if } p = 0, \\ \psi(G, \varphi(G, (p, r))) & \text{if } p \neq 0. \end{cases}$$

Then

$$\overline{f}^G = \psi(G, (f, 0)).$$

¹This notation is meant to resemble the syntax for a list in Haskell.

4.3 Buchberger's Algorithm

Continuing with our list of notational conventions from the previous section, we add a few more:

- If $X \in [T]$ and $t \in T$, then $X + t$ is the list X with t concatenated onto the end.
- If $X \in [T]$ and $Y \in [T]$, then $X + Y$ is the list X with all elements of Y concatenated onto the end in their given order.
- For $y \in T$ and $X \in [T]$, the product $y \times X \in [T \times T]$ is the list of tuples given by $((y, x) \mid x \in X)$ where we maintain the order in X .
- For $X, Y \in [T]$, the product $X \times Y \in [T \times T]$ is the list of tuples given by $((x, y) \mid x \in X, y \in Y)$ in some order.
- For a tuple $x = (f, g) \in \mathbb{F}[\mathbf{x}] \times \mathbb{F}[\mathbf{x}]$, we denote the S -polynomial $S(f, g)$ as Sx .

Let $\varphi : [\mathbb{F}[\mathbf{x}] \times \mathbb{F}[\mathbf{x}]] \times [\mathbb{F}[\mathbf{x}]] \rightarrow [\mathbb{F}[\mathbf{x}]]$ be the function defined as

$$\varphi(X, G) = \begin{cases} G & \text{if } X = \emptyset, \\ \varphi(X \setminus x_1, G) & \text{if } X \neq \emptyset \text{ and } \overline{Sx_1}^G = 0, \\ \varphi(X \setminus x_1 + (\overline{Sx_1}^G \times G), G + \overline{Sx_1}^G) & \text{if } X \neq \emptyset \text{ and } \overline{Sx_1}^G \neq 0. \end{cases}$$

Then the function $\text{gb} : [\mathbb{F}[\mathbf{x}]] \rightarrow [\mathbb{F}[\mathbf{x}]]$ defined as

$$\text{gb}(F) = \varphi(F \times F, F)$$

maps a list of polynomials f_1, \dots, f_t to a Gröbner basis for $\langle f_1, \dots, f_t \rangle$.

4.4 Efficiency

Chapter 5: Conclusion

Bibliography

- [1] *Hackage: The haskell package repository*, <https://hackage.haskell.org/package/base-4.17.0.0/docs/Prelude.html>.
- [2] Wilhelm Ackermann and David Hilbert, *Principles of Mathematical Logic*, American Mathematical Society, 1928.
- [3] Paolo Aluffi, *Algebra: Chapter 0*, first ed., American Mathematical Society, 2009.
- [4] Robert J. Bond and William J. Keane, *An Introduction to Abstract Mathematics*, first ed., Waveland Press, 1999.
- [5] Bruno Buchberger, *An algorithm for finding a basis for the residue class ring of a zero-dimensional polynomial ideal*, Ph.D. thesis, Ph. D. thesis, University of Innsbruck, Austria, 1965.
- [6] Alonzo Church, *A note on the entscheidungsproblem*, The Journal of Symbolic Logic **1** (1936), 40–41.
- [7] ———, *A formulation of the simple theory of types*, The Journal of Symbolic Logic **5** (1940), 56–68.
- [8] David A. Cox, John Little, and Donal O’Shea, *Ideals, Varieties, and Algorithms*, fourth ed., Springer, 2015.
- [9] Richard Eisenberg, *System *fc*, as implemented in *ghc**, <https://gitlab.haskell.org/ghc/ghc/-/blob/master/docs/core-spec/core-spec.pdf>.
- [10] Daniel Grayson and Michael Stillman, *Macaulay2, a software system for research in algebraic geometry*, <https://math.uiuc.edu/Macaulay2>.

- [11] David Hilbert, *Ueber die theorie der algebraischen formen*, Mathematische Annalen **36** (1890), 473–534.
- [12] Stephen Kleene, *Introduction to Metamathematics*, North-Holland, 1952.
- [13] Saunders Mac Lane, *The PNAS way back then*, The Proceedings of the National Academy of Sciences **94** (1997), 5983–5985.
- [14] Raphael Poss, *Rust for functional programmers*, arXiv preprint arXiv:1407.5670 (2014).
- [15] Emily Riehl, *Category Theory in Context*, first ed., Courier Dover, 2016.
- [16] The Sage Developers, *Sagemath, the Sage Mathematics Software System*, <https://www.sagemath.org>.
- [17] Alan Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London Mathematical Society **42** (1937), 230–265.

Tommy Meek

tommymeek123@gmail.com (828) 508-3433
<https://www.linkedin.com/in/tommymeek123/>
<https://github.com/tommymeek123>

EDUCATION

- **Master of Science, Mathematics** | GPA: 3.88 Winston-Salem, NC
Wake Forest University May 2023
- **Bachelor of Science, Mathematics, Computer Science** | GPA: 3.92 Cullowhee, NC
Western Carolina University May 2021
- **Associate of Science** | GPA: 4.00 Sylva, NC
Southwestern Community College December 2018

TECHNICAL SKILLS

- **Proficient with:** Python, Java, JavaScript, Haskell, MATLAB, Git, \LaTeX
- **Familiar with:** C, C++, MIPS, Rust, HTML, CSS, SQL, R, Bash, PowerShell, Vim, Macaulay2

PROFESSIONAL EXPERIENCE

- **Teaching Assistant** | Wake Forest University August 2021 - May 2023
 - Led weekly study sessions to ensure student success and taught courses in the absence of the professor.
 - Tutored undergraduate students in a one-on-one setting and in groups as large as twenty.
 - Graded up to fifty student assignments per week while maintaining a full-time graduate course load.
- **Math/Computer Science Tutor** | Western Carolina University August 2019 - July 2021
 - Voted *Most Valuable Tutor* by coworkers and peers.
 - Tutored undergraduate students in a one-on-one setting and in groups as large as fifteen.
 - Mentored up to thirty walk-in clients per day for all math and computer science courses.
 - Assisted in the instruction of computer science lab courses of up to forty students by answering questions and verifying the accuracy of their Python or Java lab assignments.
- **Table Games Dealer** | Harrah's Cherokee Casino September 2009 - July 2019
 - Proficiently dealt casino games including craps and blackjack at a high pace.
 - Provided service and entertainment to up to hundreds of guests per day.
 - Ensured game security by being mindful of all active players and other nearby guests.

RESEARCH

- Master's thesis: "A Functional Computer Algebra System for Polynomials"
- "An Axiomatic and Contextual Review of the Armitage and Doll Model of Carcinogenesis"
Published in *Spora: A Journal of Biomathematics* Volume 8 (2022)
- "Algebraic Properties of a Hypergraph Lifting Map"
Published in *Integers* Volume 21 (2021)
- "Avoiding Monochromatic Sub-paths in Uniform Hypergraph Paths and Cycles"
Available at <https://arxiv.org/abs/2003.00035>

PRESENTATIONS

UNCG Regional Mathematics and Statistics Conference (Virtual)

Avoiding Blue Edges in 3-uniform Hyperpaths

University of North Carolina Greensboro | Greensboro, NC | November 14, 2020

WCU SURP Symposium (Virtual)

Algebraic Properties of a Hypergraph Lifting Map

Western Carolina University | Cullowhee, NC | October 16, 2020

SCC Lunch and Learn

Canceled due to COVID-19

Undergraduate Research at Western Carolina University

Southwestern Community College | Sylva, NC | March 20, 2020

Mathematical Association of America Southeastern Section Meeting

Canceled due to COVID-19

Avoiding Blue Edges in 3-uniform Hyperpaths

High Point University | High Point, NC | March 13, 2020

Joint Mathematics Meetings Undergraduate Student Poster Session

Monochromatic Subhypergraphs in Stochastic Processes on Hypergraphs

Colorado Convention Center | Denver, CO | January 17, 2020

AWARDS AND ACHIEVEMENTS

- 2020-2021 WCU Senior Mathematics Award
- 2020-2021 WCU Advanced Computer Science Award
- 2020-2021 WCU Dean's Outstanding Scholar in Mathematics
- Score of 10 on the 2020 William Lowell Putnam Mathematical Competition

GRANTS AND SCHOLARSHIPS

- 2022 Wake Forest summer thesis grant \$3,000.00
- 2020-2021 L.E.A.R.N. Scholarship \$2,000.00
- 2020-2021 Breitenbach Scholarship \$1,381.00
- 2020-2021 George A. Milton Scholarship \$1,213.00
- 2020 SURP research grant \$5,300.00
- 2019-2020 CURM research grant \$3,000.00