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, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer 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

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 [20], 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 [16] 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 Haskeller[1] 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

[1]The 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 monoid[2], 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

[2]Recall 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 $x_1, \ldots, x_n$ 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 $\alpha_1, \ldots, \alpha_n$ are nonnegative integers. The **multidegree** of this monomial is the n-tuple $(\alpha_1, \alpha_2, \ldots) \in \mathbb{Z}_{\geq 0}^n$. The **total degree** of this monomial is the sum $\alpha_1 + \cdots + \alpha_n$.

We can simplify the notation for monomials as follows: let $\alpha = (\alpha_1, \ldots, \alpha_n)$ 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 $\alpha = (0, \ldots, 0)$, 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 $x_1, \ldots, x_n$ 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 $\alpha = (\alpha_1, \ldots, \alpha_n)$. The set of all polynomials in $x_1, \ldots, x_n$ with coefficients in k is denoted $k[x_1, \ldots, x_n]$.

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

(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 $k[x_1, \ldots, x_n]$ 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

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

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 $k[x_1, \ldots, x_n]$ is a relation $>$ on the set of monomials x^α, $\alpha \in \mathbb{Z}^n_{\geq 0}$ satisfying:

(i) $>$ is a total ordering[3].

(ii) If $x^\alpha > x^\beta$ and $\gamma \in \mathbb{Z}^n_{\geq 0}$, 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 $\alpha \in \mathbb{Z}^n_{\geq 0}$.*

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.

[3]Recall $>$ 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 $k[x_1, \ldots, x_n]$ and let $>$ be a monomial order.

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

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

(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 $\alpha = (\alpha_1, \ldots, \alpha_n)$ and $\beta = (\beta_1, \ldots, \beta_n)$ be in $\mathbb{Z}^n_{\geq 0}$. We say $x^\alpha >_{Lex} x^\beta$ 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 $x_1 > x_2 > x_3 > \ldots$.

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

Definition 8 (Graded Lex Order). Let $\alpha, \beta \in \mathbb{Z}^n_{\geq 0}$. 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 $\alpha, \beta \in \mathbb{Z}^n_{\geq 0}$. 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 $k[x_1, \ldots, x_n]$

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 $I \in k[x_1, \ldots, x_n]$ is said to be generated by a set B of polynomials if every $f \in I$ may be expressed as $f = b_1 f_1 + \cdots + b_t f_t$ for some $b_1, \ldots, b_t \in B$ and some $f_1, \ldots, f_t \in k[x_1, \ldots, x_n]$. 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 $\langle x, y \rangle \in k[x, y, z]$. 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 $I \subseteq k[x_1, \ldots, x_n]$ be an ideal other than $\{0\}$, and fix a monomial ordering on $k[x_1, \ldots, x_n]$. 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 $I \subseteq k[x_1, \ldots, x_n]$ is an ideal then by Hilbert's Basis Theorem, there is some finite set of polynomials $\{b_1, \ldots, b_t\} \subseteq k[x_1, \ldots, x_n]$ such that $I = \langle b_1, \ldots, b_t \rangle$. Since $b_1, \ldots, b_t \in I$, it is clear that $\langle \text{LT}(b_1), \ldots, \text{LT}(b_t) \rangle \subseteq \langle \text{LT}(I) \rangle$. However, this containment may be proper. Only in a very special case do we achieve equality of these two sets.

Example 11. Let $I = \langle f_1, f_2 \rangle \subseteq \mathbb{Q}[x, y]$, where $f_1 = x^3 - 2xy$ and $f_2 = x^2y - 2y^2 + x$, and use the Graded Reverse Lex order on monomials in $\mathbb{Q}[x, y]$. Then

$$x \cdot f_2 - y \cdot f_1 = x^2,$$

so that $x^2 \in I$. Thus $x^2 = \text{LT}(x^2) \in \langle \text{LT}(I) \rangle$. However x^2 is not divisible by $\text{LT}(f_1)$ or $\text{LT}(f_2)$ so $x^2 \notin \langle \text{LT}(f_1), \text{LT}(f_2) \rangle$.

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 preform long division, the quotient and remainder may depend on the order of the dividends. 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 12. Fix a monomial order on the polynomial ring $k[x_1, \ldots, x_n]$. A finite subset $G = \{g_1, \ldots, g_t\}$ of an ideal $I \subseteq k[x_1, \ldots, x_n]$ different from $\{0\}$ is said to be a **Gröbner basis** if

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

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 $G = \{g_1, \ldots, g_t\}$ 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 13. *Let $G = \{g_1, \ldots, g_t\}$ be a Gröbner basis for an ideal $I \subseteq k[x_1, \ldots, x_n]$ and let $f \in k[x_1, \ldots, x_n]$. 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 14. We will write $\overline{f}^F$ for the remainder on division of f by the ordered s-tuple $F = (f_1, \ldots, f_s)$. If F is a Gröbner basis for $\langle f_1, \ldots, f_s \rangle$, 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 $\langle f_1, \ldots, f_t \rangle$. The idea behind this algorithm is to start with a list $(f_1, \ldots, f_t)$, determine all polynomials in $\langle f_1, \ldots, f_t \rangle$ 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 15. Let $f, g \in k[x_1, \ldots, x_n]$ be nonzero polynomials.

- If $\text{multideg}(f) = \alpha$ and $\text{multideg}(g) = \beta$, then let $\gamma = (\gamma_1, \ldots, \gamma_n)$, 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 16 (Buchberger's Criterion)**.** *Let I be a polynomial ideal. Then a basis $G = \{g_1, \ldots, g_t\}$ 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 look no 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 writing high-quality software in the Haskell language. The remaining definitions in this chapter are largely taken from Emily Riehl's *Category Theory in Context* [17].

Definition 17. A **category** 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 $\mathrm{Hom}(a, b)$. If this collection is a set[4], we call $\mathrm{Hom}(a, b)$ the **hom set** of (a, b).

[4]The axioms of set theory may prevent this.

- Each object x has a designated **identity morphism** $1_x \in \mathrm{Hom}(x, x)$.
- If $f \in \mathrm{Hom}(a, b)$ and $g \in \mathrm{Hom}(b, c)$, then there is a morphism $gf \in \mathrm{Hom}(a, c)$. In this case, we say f and g are **composable** and gf is their **composite**.

This data is subject to the following two axioms:

- For any $f \in \mathrm{Hom}(a, b)$, 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 18. 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 $\mathrm{Hom}(a, b)$ be the singleton consisting of the element $(a, b) \in S \times S$. Otherwise, let $\mathrm{Hom}(a, b) = \emptyset$. 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, $(a, a) \in \mathrm{Hom}(a, a)$ and

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

so (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 19. A **functor** $\mathcal{F} : \mathsf{C} \to \mathsf{D}$ between categories C and D consists of the following data:

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

These assignments are required to satisfy the following axioms:

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

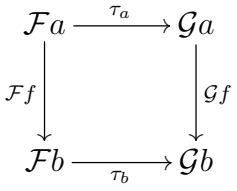
Given functors $\mathcal{F} : \mathsf{C} \to \mathsf{D}$ and $\mathcal{G} : \mathsf{D} \to \mathsf{E}$ it is easy to show that the composite $\mathcal{G}\mathcal{F}$ is a functor.

Given categories C and D, we define D^{C} to be the **functor category** from C to D. The objects in D^{C} are functors and the morphisms in D^{C} are called *natural transformations*. We call a functor between a category and itself an **endofunctor**. That is, an endofunctor on C is an object in C^{C}.

Definition 20. Given functors $\mathcal{F}, \mathcal{G} : \mathsf{C} \to \mathsf{D}$, a **natural transformation** $\tau : \mathcal{F} \Rightarrow \mathcal{G}$ is a map from objects in C to morphisms in D that satisfies the following:

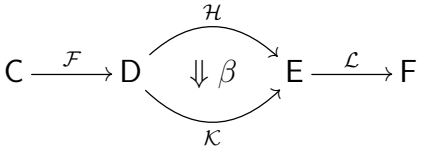
- If x is an object in C, then τ_x is a morphism in $\mathrm{Hom}(\mathcal{F}x, \mathcal{G}x)$.

- The following diagram commutes for all $a, b \in \mathsf{C}$ and all $f \in \mathrm{Hom}(a, b)$.



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.

Definition 21. Given a natural transformation $\beta : \mathcal{H} \to \mathcal{K}$ and functors $\mathcal{F}$ and $\mathcal{L}$ as displayed in



define a transformation $\mathcal{L}\beta\mathcal{F} : \mathcal{L}\mathcal{H}\mathcal{F} \Rightarrow \mathcal{L}\mathcal{K}\mathcal{F}$ by $(\mathcal{L}\beta\mathcal{F})_x = \mathcal{L}\beta_{\mathcal{F}_x}$. This is the **whiskered composite** of β with $\mathcal{L}$ and $\mathcal{F}$. It can be shown that $\mathcal{L}\beta\mathcal{F}$ is natural. If $\mathcal{L}$ is the identity functor, then $\mathcal{L}\beta\mathcal{F} = \beta\mathcal{F}$. If $\mathcal{F}$ is the identity functor, then $\mathcal{L}\beta\mathcal{F} = \mathcal{L}\beta$.

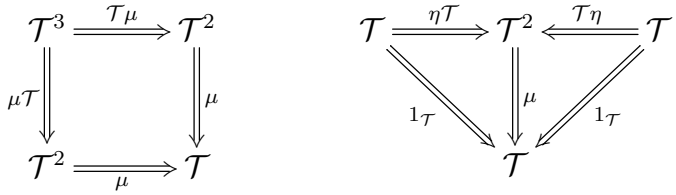
We won't spend much time discussing the finer points of category theory. If teaching the fundamentals to a class of advanced undergraduate mathematics students over the course of a semester in their senior year is a bold undertaking, then explaining it in a single section of this document is a Kobayashi Maru[5]. We do however need to address one more categorical construction before proceeding.

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

Definition 22. A **monad** on a category C consists of

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

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



In these diagrams, $\mathcal{T}\eta, \eta\mathcal{T} : T \Rightarrow T^2$ are defined as

$$(\mathcal{T}\eta)_x = \mathcal{T}(\eta_x) \qquad \text{and} \qquad (\eta\mathcal{T})_x = \eta_{\mathcal{T}x},$$

and $\mathcal{T}\mu, \mu\mathcal{T} : T^3 \Rightarrow T^2$ are defined as

$$(\mathcal{T}\mu)_x = \mathcal{T}(\mu_x) \qquad \text{and} \qquad (\mu\mathcal{T})_x = \mu_{\mathcal{T}x}.$$

The reason for this definition will become apparent in Chapter 3.

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 $f : X \to Y$ 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 -> 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 add(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 $f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. In Haskell this is accomplished with tuples.

```
add :: (Int,Int) -> Int
add (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 $\mathbb{Z} \to \mathbb{Z}$. This is the more common way to implement such a function in Haskell.

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

The strength of this approach is the ability to partially evaluate functions. If we want a function $\mathbb{Z} \to \mathbb{Z}$ that returns the sum of a number and five, we write `add 5`. So `add` is a function that takes an integer and returns a function and `add 5` is a function that takes an integer and returns an integer. We say that `add 5` is a *partially evaluated* function. We say that an expression like `add 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 recursion[1].

```
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[1].

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 [21]. 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 Data Types extension. When used in conjunction with `DataKinds`, 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 may annotate the value constructors with their type, and we may 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. An example is the `Show` typeclass which is for types that can easily be converted to strings. The `Show` typeclass is defined like this:

```
class Show a where
    show :: a -> String
```

The keyword `class` means that we're about to define a new typeclass. After the `where` keyword, we list all the functions that are specified by the typeclass. To make some type an instance of `Show`, we need to define the `show` function, which takes a term of that type and outputs a string. If we wanted to make `Color` an instance of the `Show` typeclass, we could implement it like this:

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

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. This is why our implementation of the `Vector n` type is so powerful. Since the type system is doing the work of ensuring type safety, our code can be cleaner, safer, and more efficient.

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.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Because of currying, we can rewrite the signature of `fmap` as

```
fmap :: (a -> b) -> (f a -> f b)
```

Let's see how `Maybe` is a functor.

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

Mapping any function over `Nothing` results in `Nothing`, while mapping a function over `Just` something applies that function to the contents of the `Just`.

It is not uncommon in Haskell for typeclasses to come equipped with recommended rules for how they should work. These rules are not enforced by the compiler, but should be followed whenever possible. If a type is an instance of a typeclass with these rules but does not follow them, it should be clearly documented with the reason why. In the case of `Functor`, there are two of these rules known as the *functor laws* that every `Functor` instance should obey.

```
fmap id == id
fmap (f . g) == fmap f . fmap g
```

The first law states that if we map the `id` function over a functor, the functor that we get back should be the same as the original functor. The second law says that composing two functions and then mapping the resulting function over a functor should be the same as first mapping one function over the functor and then mapping the other one [14]. It is easily verified that the implementation of `Maybe` obeys the functor laws.

So an instance of `Functor` consists of two things:

- a type constructor that maps a type `a` to another type `f a`, and
- a way to map functions of type `a -> b` to functions of type `f a -> f b`,

and to be a true functor, the data above must preserve the identity and preserve function composition. If this does not sound familiar to the reader, then perhaps

they should review the last section of Chapter 2. The `Functor` typeclass seems to be specifying that the type constructor and `fmap` functions of a type form a functor between some categories. Indeed, this is where `Functor` gets its name. But what categories are we dealing with? The type constructor is a map from a Haskell type to another Haskell type and `fmap` is a map from Haskell functions to other Haskell functions. Do these form a category though? Could we build a category where the objects are Haskell types and the morphisms are the functions between those types? It turns out that we cannot. Not completely anyway. The majority of the problems come from `undefined` values and functions that do not terminate. These are certainly a part of Haskell and must be accounted for. However, if we ignore these complications as well as a few other problems, then we may proceed with defining a category.

Definition 23. The category Hask has the types of Haskell as its objects. For types t and s, the set $\mathrm{Hom}(t, s)$ consists of the functions from t to s.

As previously mentioned, the above definition is not entirely accurate. For the purposes of this discussion however, we may assume that this definition holds. For a more thorough treatment of the subject including a list of problems with this definition and ways to get around them, see [15]. Using this definition, an instance of `Functor` is an endofunctor on the category Hask.

A special type of functor in this category is an *applicative functor*. We will not go into any detail regarding the implementation or theory of applicative functors except for the following brief definition and description.

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

The `pure` function is a special way to map a regular term to a functorial term without doing anything else. The `<*>` operator is a way to map a functorial function to a function of functorial terms. Note that in Haskell syntax, when defining a function

with non-alphanumeric characters surrounded by parentheses, that function is used infix and we call it an operator. Sticking with our running example, lets see how the `Maybe` functor is applicative.

```
instance Applicative Maybe where
    pure          = Just
    Just f  <*> m = fmap f m
    Nothing <*> m = Nothing
```

There are several *applicative laws* that should be satisfied in addition to the functor laws.

```
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

In the above, `(.)` is function composition and `($)` is function application. For an explanation of these laws, see the documentation [1]. The reason for defining applicative functors here is so that we may discuss a very special applicative functor with a familiar name.

We ended Chapter 2 by introducing an odd categorical construction: an endofunctor $\mathcal{T} : \mathsf{C} \to \mathsf{C}$ equipped with two natural transformations $\eta : 1_{\mathsf{C}} \Rightarrow \mathcal{T}$ and $\mu : \mathcal{T}^2 \Rightarrow \mathcal{T}$ that satisfied a pair of commutative diagrams. We called this structure a *monad*. Monads are important enough in their native domain of category theory, but where they really shine is in the realm of functional programming. In Haskell, `Monad` is a typeclass that may be implemented on an applicative functor. Instances of `Monad` are monads on the category Hask.

```
class Applicative m => Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a
    m >> k  = m >>= \x -> k
    return  = pure
```

The `Monad` typeclass specifies three functions, `>>=`, `>>`, and `return`. Note that two of these functions are defined in the `class` definition. Those are default implementations.

They may be overridden, but in the case of `Monad`, they rarely are. The `return` function is equivalent to `pure`. The `>>` function is rarely used, so we will not discuss it here. The important one of this bunch is the one that does not have a default implementation, `>>=`. This is called the *bind* operator.

In a less abstract sense, a monad, and to a lesser extent, any functor, may be thought of as a *computational context*. The bind operator takes a value of type `a` that is already in some context `m` and a function of type `a -> m b` that maps standard values to contextualized values. Its output is the image of the decontextualized value under that function. The context here could be almost anything. Lists are monads whose computational context is that there are more than one terms of a given type. The `Maybe` functor is a monad where the computational context is that any term may or may not exist. When Haskellers are writing software, it is usually more insightful to think of monads this way than as endofunctors and natural transformations on the category Hask.

Since all monads are applicative functors and all applicative functors are functors, monads should obey the functor laws and the applicative laws. In addition, there are three more laws a monad should obey. These are the *monad laws*.

```
return x >>= f = f x
m >>= return = m
m >>= (\x -> f x >>= g) = (m >>= f) >>= g
```

The first monad law states that if we take a value, put it in a default context with `return` and then feed it to a function by using bind, it's the same as just taking the value and applying the function to it. The second law states that if we have a monadic value and we use bind to feed it to `return`, the result is our original monadic value. The final monad law says that when we have a chain of monadic function applications with bind, it shouldn't matter how they're nested [14].

We mentioned that `Maybe` is a monad. Let's take a look at its implementation.

```
instance Monad Maybe where
    (Just x) >>= f = f x
    Nothing  >>= f = Nothing
```

In the above, `f` is a function of type `a -> Maybe b`. We can see that binding `f` to `Just x` is simply `f x` where `x` has type `a` and `f x` has type `Maybe b`. Binding `f` to `Nothing` is unsurprisingly `Nothing`.

So mathematically, a monad is an endofunctor on Hask along with two natural transformations making the diagrams we saw at the end of Chater 2 commute. Programatically, a monad is an applicative functor with a `return` function and a bind function that satisfy the monadic laws in addition the the functor and applicative laws, but are these two notions the same? To justify naming this typeclass `Monad`, there must be some relationship.

The `return` function is just the unit natural transformation η, but what about bind? Is bind the multiplication natural transformation μ that we saw before? Perhaps surprisingly, it is not. There is another function that instances of the `Monad` typeclass have access to: the `join` function.

```
join :: (Monad m) => m (m a) -> m a
join x = x >>= id
```

It turns out one could define a monad using `join` instead of bind and then define bind using `join`. This method is equivalent to the standard approach. In fact, the developers of Haskell tried fairly recently to include `join` in the definition for `Monad` and allow developers the option of defining either `join` or bind when creating a `Monad` instance. Unfortunately, due to some technical limitations, this is not currently possible, though there have been more recent attempts to fix this problem [18]. If this technical problem were overcome, we could redefine the `Monad` typeclass as

```
class Applicative m => Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    join    :: (Monad m) => m (m a) -> m a
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a
```

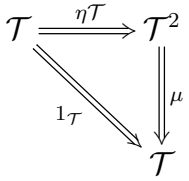
```
    x >>= f = join (fmap f x)
    join x  = x >>= id
    m >> k  = m >>= \x -> k
    return  = pure
```

In this definition, since bind and `join` are each defined in terms of the other, a user need only define one of them and they get the other for free.

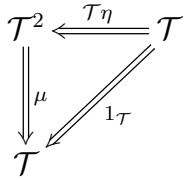
We can also reformulate our monad laws using `join` and `fmap` instead of bind.

```
join . return = id
join . fmap return = id
join . fmap join = join . join
```

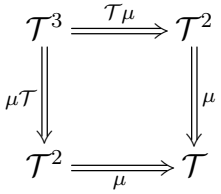
In categorical terms, `join` is the μ transformation we wanted. For a functor $\mathcal{T}$ and a type `a`, μ gives us a function from the type $\mathcal{T}^2 a$ to the type $\mathcal{T} a$. The first monad law expressed with `join` says that the following diagram commutes:



The second monad law says that this diagram commutes:



The third monad law says that this final diagram commutes:



So using `join` to define a monad instead of bind makes the parallels to category theory much more explicit. To complete our running example, let's see how `Maybe` would be defined with `join` instead of bind.

```
instance Monad Maybe where
    join (Just (Just x)) = Just x
    join _ = Nothing
```

The _ is a place holder. It means that any expression that doesn't look like the first one will evaluate to `Nothing`.

A `Monad` comes equipped with three functions. `return` puts an expression into some computational context, `join` reduces the nested level of context by one, and bind is a mixture of both that ends up being quite useful in practice. We mentioned before that lists as well as `Maybe` functors are monads, but there are many more examples, like `Either`, `State`, and `IO`. That's right. Any `IO` operation in Haskell is a monad, even printing to the screen.

Encapsulating all computational contexts in monads makes the codebase safer and cleaner and encourages composition in a way that imperative programming can't compete with. However, this method of doing things is quite unfamiliar to most programmers. That is one of the reasons it is so important to explore functional solutions to problems where currently only imperative solutions exist. In the next chapter, we will discuss one such solution.

Chapter 4: The Polynomial-Algorithms Package

In this section, we will discuss how the `polynomial-algorithms` package was implemented. We will take a bottom-up approach, starting with the lower-level modules that define the more basal structures and working our way up to the higher-level modules and user facing functions. The primary data structure in this package is the `Polynomial` type, which is built on top of a `Monomial` type and a `Coefficient` type. This implementation allows for modularity. We may change the implementation of `Monomial` or `Coefficient` without changing the `Polynomial`.

The primary purpose of these types is their use in the `Algorithms` module. This module contains functions that carry out recursive versions of several algorithms related to polynomials. Polynomial long division and Buchberger's algorithm are the primary features of this module, though it is possible to use the existing `Polynomial` type to implement more algorithms in the future.

4.1 The Polynomial Type

One very simple module is `RingParams`. It merely contains enums that specify the rings and monomial orderings that are allowed throughout the package. Its purpose is to reduce dependencies between the larger modules. The entirety of its content is the following.

```
data Ring = Q | Zero | FTwo | FThree | FFive deriving (Eq,Read,Show)

data MonOrder = Lex | GLex | GRevLex deriving (Eq,Read,Show)
```

Note that the `Eq` typeclass allows a type to use the `==` operator and the `Read` typeclass allows a type to be read from a string. Henceforth, the abbreviation `RP` stands for `RingParams` so if the expression `RP.Q` appears, it is referring to the value constructor

`Q` from this module. Additionally for this section, we will omit class constraints from function definitions for brevity.

The first module of real substance is `Coefficient` which contains the `Coefficient` type. This type is defined as follows.

```
data Coefficient :: RP.Ring -> * where
    Q       :: Rational -> Coefficient r
    Zero    :: Coefficient r
    FTwo    :: Integer -> Coefficient r
    FThree  :: Integer -> Coefficient r
    FFive   :: Integer -> Coefficient r
```

Note that `Rational` is a built in type that represents rational numbers. Since we are using the `DataKinds` extension, Our definition of `Ring` in `RingParams` created not only the type `Ring`, but also the kind `Ring`. The top line of this definition says that `Coefficient` is a type of kind `RP.Ring -> *`. That is, it is a type constructor whose input is a type of kind `RP.Ring` and whose output is a type of kind `*`. We have defined five value constructors for this type. These value constructors are not the same as the ones we saw in the `RingParams` module. Those were merely enums. These are the actual types that the `Polynomial` type will use. The first value constructor is `Q` and represents the field of rational numbers. The next represents the zero ring. The other three represent finite fields with two, three, and five elements respectively. When using any of the features of this package, we may choose which of these fields we wish to use. That means we can find a Gröbner basis for a list of polynomials over the rationals and then find a Gröbner basis for the same list over $\mathbb{F}_2$.

Note that we use the type synonym `type Q = Coefficient RP.Q` throughout this package for brevity. Type synonyms are simply aliases to keep the code more concise. The most interesting part of the `Coefficient` module other than the type definitions is the implementation of typeclasses. The first typeclass we will explore is `Num`. This typeclass specifies that a type should behave like an algebraic ring. This means that this typeclass will have functions that are allowed in a ring.

```
instance Num Q where
    (Q r) + (Q s) = Q (r + s)
    (Q r) - (Q s) = Q (r - s)
    (Q r) * (Q s) = Q (r * s)
    abs (Q r)     = Q (abs r)
    signum (Q r)  = Q (signum r)
    fromInteger n = Q $ n % 1
```

The `abs` and `signum` functions define the absolute value and sign of a term respecitvely. It is a bit unfortunate that these are part of the `Num` typeclass as they are not inherent properties of a ring, but as they are, we must define them. This implementation is actually quite straightforward because `Rational` is already an instance of the `Num` typeclass. The only reason we needed to define `Q` at all is so that it would fit in with the other coefficient rings. Note that `%` is a value constructor for the `Rational` type. Next, lets examine how `FFive` is a `Num` instance.

```
instance Num FFive where
    (FFive n) + (FFive m)   = FFive ((n + m) `mod` 5)
    (FFive n) - (FFive m)   = FFive ((n - m) `mod` 5)
    (FFive n) * (FFive m)   = FFive ((n * m) `mod` 5)
    abs (FFive n)           = error "No absolute value in F5."
    signum (FFive n)        = error "No sign in F5."
    fromInteger n           = FFive $ n `mod` 5
```

As we can see, this is once again rather straightforward except that we are performing all operations modulo 5. The rest of the rings here have similar `Num` instances.

The other typeclass worth mentioning in the `Coefficient` module is `Fractional`. This typeclass specifies multiplicative inverses for a type that is already an instance of `Num`. An instance of `Fractional` is usually a division ring.

```
instance Fractional Q where
    recip (Q r)  = Q (denominator r % numerator r)
    fromRational = Q
```

The `recip` function specifies the multiplicative inverse of a term in a `Fractional` type. Because `Rational` is already `Fractional`, this instance of `Q` is simple. The `numerator` and `denominator` functions give the numerator and denominator of a `Rational` respectively. We must take some care in making `Coefficient` an instance of `Fractional`

using the `FFive` value constructor because this constructor is based on an `Integer`, not a `Rational`, and `Integer` is not already `Fractional`.

```
instance Fractional FFive where
    recip 0        = error "Cannot divide by zero."
    recip 1        = FFive 1
    recip 2        = FFive 3
    recip 3        = FFive 2
    recip 4        = FFive 4
    fromRational n = p * recip q where
        p = fromInteger (numerator n)
        q = fromInteger (denominator n)
```

Here, we have defined each inverse explicitly. The other way to do this would be to use the Euclidean algorithm, but when the field characteristic is this low, it is clearer to do it this way.

The next type we will discuss is `Monomial`. There are actually two separate implementations of this type. One implementation is contained in the `DenseMonom` module and is based on the `fixed-vector` package [19]. Using this implementation, the `Monomial` type is defined as

```
newtype Monomial :: Nat -> RP.MonOrder -> * where
    MakeMon :: { degVec :: Vec n Int } -> Monomial n o
```

This definition uses the `newtype` keyword, which is the same as `data` but more efficient in some cases and `Vec` is just the type we're using from `fixed-vector`. It also uses *record syntax* which means that we automatically get an accessor function `degVec` that gives us the multidegree of a `Monomial` stored as a fixed length vector. Observe that the first parameter of `Monomial` is of kind `Nat`. This enables us to make the number of variables in our polynomial ring part of the type. For a discussion of why this is important, see the end of section 3.1.

In Haskell, `Map k v` is a type that represents key-value pairs. The keys are of type `k` and the values are of type `v`. An `IntMap a` is a `Map` whose keys are of type `Int` and whose values are of type `a`. The other implementation of `Monomial` uses an `IntMap` to store the exponents instead of a `Vec`. This implementation is in the `SparseMonom`

module. We call it sparse because if we are in a polynomial ring with thirty variables, then using `DenseMonom`, the monomial x^2 will be stored as a vector with twenty-nine 0's and a single 2, but in `SparseMonom`, it will be an `IntMap` with a single key-value pair. In this module, we define `Monomial` as follows.

```
newtype Monomial :: Nat -> RP.MonOrder -> * where
    MakeMon :: { degMap :: IntMap Int } -> Monomial n o
```

This definition is similar to before, but with `degMap` in place of `degVec`.

We mentioned in Chapter 2 that Haskell used more mathematical terminology than most languages. The next two typeclasses are good examples of that.

```
class Semigroup a where
    (<>) :: a -> a -> a

class Semigroup a => Monoid a where
    mempty :: a
```

As the names imply, an instance of the `Semigroup` typeclass should be a semigroup, and an instance of the `Monoid` typeclass should be a monoid. The `<>` operator is the semigroup operation, and the `mempty` function is the identity in the monoid. The `Monomial` type is both a semigroup and a monoid. In the dense representation, this is implemented as

```
instance Semigroup (Mon n o) where
    a <> b = MakeMon $ zipWith (+) (degVec a) (degVec b)

instance Monoid (Mon n o) where
    mempty = MakeMon $ fromList $ replicate nn 0
        where nn = (fromInteger . reflect) (Proxy :: Proxy n)
```

The interesting part of these definitions is the `where nn` clause. This is how the `Nat` kind works. If we need to access the numeric value associated with a type-level natural number, we `reflect` the value from a `Proxy`. This can be a bit difficult to understand, but just know that this is how we actually use the information that we are storing in the type of our data. In this case, the `mempty` function needs to know the type of `Monomial` it is going to output, otherwise, it won't know how big to make the vector.

The sparse version of `Monomial` is also an instance of `Semigroup` and `Monoid`.

```
instance Semigroup (Mon n o) where
    a <> b = MakeMon $ unionWith (+) (degMap a) (degMap b)

instance Monoid (Mon n o) where
    mempty = MakeMon $ empty
```

Notice that in this case, we didn't need the type-level information to form the monoidal identity. That's a benefit of a sparse representation. The identity is just stored as an empty map.

One other important typeclass is `Ord`. This specifies the ordering of a type and how they work with the `>` and `<` operators in Haskell. In the case of `Monomial`, this of course corresponds to the monomial orderings we discussed in Chapter 2. We want to be able to run algorithms for any of the three main orderings we discussed, so `Monomial` is an instance of `Ord` with each ordering separately. In the dense implementation, this is accomplished with the following.

```
instance Ord (Mon n RP.Lex) where
    compare a b = compare (degVec a) (degVec b)

instance Ord (Mon n RP.GLex) where
    compare a b = let aVb = compare (totalDegree a) (totalDegree b)
                  in  if aVb == EQ
                      then compare (degVec a) (degVec b)
                      else aVb

instance Ord (Mon n RP.GRevLex) where
    compare a b = let aVb = compare (totalDegree a) (totalDegree b)
                      a'  = reverse $ degVec a
                      b'  = reverse $ degVec b
                  in  if aVb == EQ
                      then compare b' a'
                      else aVb
```

Notice that we are partially evaluating the `Monomial` type constructor by putting in a concrete type for the `o` parameter but leaving the `n` parameter arbitrary. We use the same strategy in `SparseMonom`, but the implementation is a little trickier. We won't go into detail here, but the lack of zero exponents poses a challenge in determining the order efficiently.

Since we have two different implementation of this type, we are able to compare their efficiency. In all of our testing, we have found that the sparse implementation is significantly faster than the dense. In the case of a polynomial ring with many variables this is to be expected, but the fact that it is faster with only two or three variables is a bit surprising. It is likely due to the fact that the `IntMap` type is much more widely used than the `Vec` type, so the community effort to make it efficient has been more successful.

Now that we have a `Coefficient` type and a `Monomial` type, we can build the `Polynomial` type. A `Polynomial` is a `Map` from `Monomial` to `Coefficient`.

```
newtype Polynomial :: RP.Ring -> Nat -> RP.MonOrder -> * where
    MakePoly :: { monMap :: Map (Mon n o) (Coef r) } -> Polynomial r n o
```

Throughout this module, `Mon` is a type synonym for `Monomial`, `Coeff` is a type synonym for `Coefficient`, and `Poly` is a type synonym for `Polynomial`. Since polynomials form a ring, it makes sense to make `Polynomial` an instance of the `Num` typeclass.

```
instance Num (Poly r n o) where
    f + g           = makePoly $ unionWith (+) (monMap f) (monMap g)
    f * g           = f `polyMult` g
    abs             = error "Polynomials do not have absolute values."
    signum _        = error "Polynomials do not have signs."
    fromInteger n   = makePoly $ singleton mempty (fromInteger n)
    negate          = makePoly . map negate . monMap
```

Polynomial addition is just unioning all the terms together and if any have the same monomial, we add their coefficients. The `makePoly` function uses the `MakePoly` value constructor but it filters out terms whose coefficients are 0. The functions `abs` and `signum` once again don't really make sense for a polynomial ring, but we must define them anyway. An integer n may be represented as a polynomial with the map from the identity monomial to n. The negative of a polynomial is given by negating each of the coefficients. The only difficult definition here is multiplication. What we have is only a call to a helper function. Lets take a look at `polyMult`.

```
f `polyMult` g = if numTerms f < numTerms g
```

```
                then f `leftPolyMult` g
                else g `leftPolyMult` f where
    f `leftPolyMult` g = foldlWithKey (distrOver g) 0 (monMap f)
    distrOver g p m c  = p + (leftMultWithCoef m c g)
```

The `leftMultWithCoef` function multiplies a single term by a polynomial. The reason for the `if` statement is efficiency. It is more efficient to multiply polynomials in this manner when the one with fewer terms is on the left. This is because of the way our `foldl` works.

The last function we will examine in the `Polynomial` module is `sPoly`. Given two polynomials f and g, `sPoly` computes $S(f, g)$ as defined in Chapter 2. One difficulty with this is that $S(f, 0)$ is undefined, so how should this function behave if the zero polynomial is one of its inputs? One way to handle potential failure in Haskell is with the `Maybe` monad. So instead of `sPoly` having an output of type `Poly r n o`, its output will be of type `Maybe (Poly r n o)`.

```
sPoly :: Poly r n o -> Poly r n o -> Maybe (Poly r n o)
sPoly f g = (-) <$> redf <*> redg where
    lcm = lcmMon <$> leadMonom f <*> leadMonom g
    fNorm = divideByLeadTerm <$> fmap asPoly lcm <*> leadTerm f >>= id
    gNorm = divideByLeadTerm <$> fmap asPoly lcm <*> leadTerm g >>= id
    redf = (*) <$> fNorm <*> Just f
    redg = (*) <$> gNorm <*> Just g
```

Recall that the S-polynomial of f and g is defined as

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

where x^γ is the least common multiple of $\text{LT}(f)$ and $\text{LT}(g)$. In the `sPoly` code above, `lcm` is x^γ, `fNorm` is $\frac{f}{\text{LT}(f)}$, `gNorm` is $\frac{g}{\text{LT}(g)}$, `redf` is $\frac{x^\gamma}{\text{LT}(f)} \cdot f$, and `redg` is $\frac{x^\gamma}{\text{LT}(g)} \cdot g$. The problem is that `leadMonom` and `leadTerm` both return `Maybe` values, because the zero polynomial has no lead term. The pattern of using `<$>` and `<*>` as above is a common idiom in Haskell for dealing with monads, especially the `Maybe` monad. If I write a function that takes input of type `Mon n o` and I give it a `Maybe (Mon n o)`, then it

won't work. So I could either make sure that all of my terms are `Maybe` values all the time or I could write two versions of every function, one for regular terms and one for monadic terms. Neither of these options are acceptable, so instead we use this pattern. Doing this makes it so that if a `Nothing` value is encountered, it is passed along until it is dealt with. If all of the values are `Just` values, then it evaluates the function as normal. It may look confusing if you're not used to the syntax, but it is a great way to handle potential failure in functional programming.

Now that we have a type that represents polynomials, how do we use it? How do we create a polynomial of in a polynomial ring over $\mathbb{Q}$ with thee variables using the lexicographical monomial order for example? Since these parameters are type-level information, we cannot pass term-level parameters to our constructor functions. The answer is quite elegant. We simply use a type annotation.

```
type R = Polynomial RP.Q 3 RP.Lex
f = (fromString "x^5 + y^4 + z^3 - 1") :: R
```

Here, we have a polynomial `f` that is a member of the polynomial ring $\mathbb{Q}[x, y, z]$ whose monomials use the lexicographical ordering. Now we can analyze `f` with the tools in the `Algorithms` module.

4.2 The Division Algorithm

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

- Given a set T, denote[1] the set of finite ordered lists of elements of T as $[T]$. Let $\emptyset \in [T]$ denote the empty list.
- For a list $X \in [T]$, denote the nth 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.

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

- Suppose n is fixed throughout and let $\mathbb{F}[\mathbf{x}] = \mathbb{F}[x_1, \ldots, x_n]$.

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

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

Let $\psi : [\mathbb{F}[\mathbf{x}]] \times (\mathbb{F}[\mathbf{x}] \times \mathbb{F}[\mathbf{x}]) \to \mathbb{F}[\mathbf{x}]$ 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

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

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 with all elements of Y concatenated onto the end of X maintaining the order of each.
- 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 $\varphi : [\mathbb{F}[\mathbf{x}] \times \mathbb{F}[\mathbf{x}]] \times [\mathbb{F}[\mathbf{x}]] \to [\mathbb{F}[\mathbf{x}]]$ be the function defined as

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

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

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

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

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] Miran Lipovača, *Learn You a Haskell for Great Good!*, No Starch Press, 2011.

[15] Bartosz Milewski, *Category Theory for Programmers*, Bartosz Milewski, 2019.

[16] Raphael Poss, *Rust for functional programmers*, arXiv preprint arXiv:1407.5670 (2014).

[17] Emily Riehl, *Category Theory in Context*, Courier Dover, 2016.

[18] Ryan Scott, *How QuantifiedConstraints can let us put join back in Monad*, https://ryanglscott.github.io/2018/03/04/how-quantifiedconstraints-can-let-us-put-join-back-in-monad/, 2018.

[19] Shimuuar, *fixed-vector*, https://github.com/Shimuuar/fixed-vector.

[20] The Sage Developers, *Sagemath, the Sage Mathematics Software System*, https://www.sagemath.org.

[21] 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