Software Specialization as Applied to Computational Algebra

Software Specialization as Applied to Computational Algebra

By
Pouya Larjani, M.Sc.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Doctor of Philosophy
Department of Computing and Software
McMaster University

DOCTOR OF PHILOSOPHY (2012) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: Software Specialization as Applied to Computational Algebra.

AUTHOR: Pouya Larjani, M.Sc. (McMaster University)

SUPERVISOR: Dr. William M. Farmer

NUMBER OF PAGES: viii, 198

Abstract

A great variety of algebraic problems can be solved using Gröbner bases, and computational commutative algebra is the branch of mathematics that focuses mainly on such problems. In this thesis we employ Buchberger's algorithm for finding Gröbner bases by tailoring specialized instances of Buchberger's algorithm via code generation. We introduce a framework for meta programming and code generation in the F# programming language that removes the abstraction overhead of generic programs and produces type-safe and syntactically valid specialized instances of generic programs. Then, we discuss the concept of modularizing and decomposing the architecture of software products through a multistage design process and define what specialization of software means in the context of producing special instances. We provide a domain-specific language for the design of flexible, customizable, multistage programs. Finally, we utilize the aforementioned techniques and framework to produce a highly parametrized, abstract and generative program that finds Gröbner bases based on Buchberger's original algorithm, which, given all the proper definitions and features of a specific problem in computational algebra, produces a specialized instance of a solver for this problem that can be shown to be correct and perform within the desired time complexity.

Acknowledgements

Many people have helped me during my graduate studies who deserve my eternal thanks. First and foremost, I would like to express my gratitude towards my advisor, Dr. William Farmer, for helping me out through the many years that I have been at McMaster and his excellent guidance. Thank you for giving me the opportunity to do this research.

I am thankful to the members of my supervisory committee for their invaluable advice and reviews of my work: Dr. Jacques Carette for giving me the base idea for this thesis with his work in the generation of Gaussian Elimination algorithms and guiding me with the meta programming and code generation aspects of my research. Dr. Jeff Zucker for all the interesting conversations we had (both on and off topic) and his interest in my research on software specializations.

I would also like to thank my family for encouraging me and their support in every step of the way. My colleagues who kept my company throughout my graduate studies and patiently listened to my attempts, ideas, and troubles that I encountered, Marc and Gord, you have my thanks.

Contents

List of Figures

Chapter 1

Introduction

Many applications of computer programs come in the form of computational algebra problems. For example, finding the inverse of a matrix, solving a Sudoku puzzle, finding the maximum flow of a network, or even proving geometric identities are all special instances of such algebraic problems. All of these sample problems as well as many other computational algebra problems fall within specific classes of problems for which there exist finely tuned, specialized algorithms for solving them. The examples given above are instances of Gaussian Elimination, Graph Colouring, Linear Programming, and Geometric Theorem Proving problems, respectively. Historically, there have been many algorithms developed to solve these problems, each with its own strengths and weaknesses [22], but these algorithms often share little in common with each other.

Each of these problems can be expressed as a statement about membership of a polynomial ideal. In fact, it is shown that any problem in computational commutative algebra — as defined by Kreuzer et al. [48] — can be expressed as a membership query within a polynomial ideal. In this case, it is sensible to attempt to solve all such problems together as a general ideal membership problem. At the heart of computational algebra lies a very specific kind of basis for a polynomial ideal called the Gröbner basis which provides a straight-forward and simple method for determining membership within a given ideal. Thus we can provide the answers to any computational commutative algebra problem by expressing the question as an ideal membership problem and then finding a Gröbner basis for this ideal.

There are thousands of such problems in computational algebra that may be solved via Gröbner bases; many of these problems have their own special algorithms, but

most of them (many of which are not even named) do not possess such fine tuned or specialized methods. The goal of this thesis is to be able to provide programs which solve any of these problems — even the future possibilities of new classes of problems — in such a way that the computed solution can be shown to be correct, that the performance of the program is acceptable for the specific problem, and that the solution is automatically generated given reasonable user input.

With such a grand vision for generating algorithms to computational commutative algebra problems, it is evident that the entire solution revolves around Gröbner bases and their computation. There are a few known methods for computing Gröbner bases for polynomial ideas, such as Buchberger's algorithm by Bruno Buchberger [12] and the F4/F5 algorithms by Jean-Charles Faugère [35, 36]. Although these algorithms on their own can be directly used to solve the given algebraic problems, they suffer from certain difficulties that render them unusable for this goal:

- The user is required to perform the encoding of the problem as a polynomial ideal and then to reinterpret the corresponding basis back into the original domain of the problem.

- The performance of these algorithms is usually unacceptable for the majority of algebraic problems due to the fact that finding Gröbner bases is a hard problem in general (doubly exponential in the worst case) [54].

- Finding the right implementation and optimization of these algorithms may require more programming and knowledge of Gröbner bases by the user than implementing the specialized algorithm for that specific problem originally required.

We seek to address these difficulties by generating specialized instances of Buchberger's algorithm according to the mathematical and behavioural properties of the specific problem defined by the user. A toolkit library of many customized and parametric implementations of each individual feature and aspect used in this algorithm is provided to ease the task of choosing the right optimized instance for the user, and a proof framework is provided in such a way that users can develop a proof of correctness for each individual generated algorithm when required.

1.1 Objectives and Components

We started with a clear research goal defined for this thesis: to develop a code generator that produces specialized instances of Gröbner bases solvers, tailored specifically for each computational algebra problem as defined by the user. Moreover, we have further refined this goal by basing the solver on Buchberger's algorithm for finding Gröbner bases. We achieve this goal via defining smaller sub-tasks and related objectives that will be addressed during this research:

(1) *Describe the mathematical background for Gröbner bases*: Carefully define the mathematical requirements for computational algebra and specifically, define the background required for expressing algebraic problems as ideal membership questions. This task also requires us to define the algebraic objects and algorithms in such a way that every element can be represented programmatically in a mechanized environment. This task is mostly based on the work by Kruezer and Robbiano [48, 49] in the CoCoA framework [3] and by Buchberger and Winkler [13] in applications of Gröbner bases.

(2) *Design a framework for code generation*: Develop a code generation framework in a programming language that supports all the features required to produce specializations of Buchberger's algorithm. The programming language we chose for this thesis is F# [85], and we develop, both formally and programmatically, a framework for code generation using the meta programming features provided by F#. Additionally, we produce a domain-specific language for the developed code generator to further advance the flexibility and the readability of the generators.

(3) *Formalize software specification and specialization*: Define the theoretical necessities and the practical machinery required to correctly produce software specializations according to the formal specifications for the software components. This task requires a careful definition of specifications in a logical framework when concerned with properties of a software system, the logical relationship of the specifications between refinements and abstractions of software, the meaning and notation for instantiating or implementing a specification as a computer program, and the formal definition of software specialization and generalization[1]. We also require the definition of a module structure which properly de-

[1] We are only concerned with specialization in the context of specification of software components.

composes the architecture of the software into specific and orthogonal modules in such a way that we can specify and specialize each module independently from the rest of the architecture. The modular breakdown is based on early work by D. L. Parnas [65].

(4) *Design the architecture for generating Gröbner bases solvers*: Decompose Buchberger's algorithm into orthogonal modules — module signatures and individually tailored modules instantiations — as per the earlier definition of a modular software design. This also requires an implementation of all the individual modules as abstract and extensible interfaces in F#, as well as the formal specifications of each module. The code generator uses specializations of these modules to generate the final instance of the algorithm according to the choice of specializations and the other properties of the solution.

(5) *Design a computational algebra library*: Develop a mathematical framework for performing computational algebra in the programming language of choice. In addition to a formal specification and a computer implementation of each algebraic object, we require the development of the code generators for instantiating and performing computations on these modules. These modules reflect all the mathematical objects defined earlier and form a basis for development of any specializations of the modules in the decomposition of Buchberger's algorithm.

(6) *Develop the code generator for Buchberger's algorithm*: Ultimately, program the code generator for instantiating specialized solvers for finding Gröbner bases in this code generation framework that uses the earlier-defined algebra libraries and is based on the modular breakdown of Buchberger's algorithm as defined earlier. This code generator must take into consideration every choice of specialization and optimization as provided by the user and produce a final program that is tailored specifically for the problem at hand. The code generator must be sufficiently general to produce both a full generic implementation of Buchberger's algorithm as well as a computationally optimized and specialized instance of this algorithm that is tailored specifically for a well defined and small computational algebra problem with a far (complexity-wise) simpler algorithm.

(7) *Generate some specialized sub-algorithms*: Finally, to demonstrate the usability and versatility of this code generator, produce and test many sample instances

of this algorithm which cover a wide range of possible specializations, optimizations, and limitations to show both the correctness and the simplicity of generating such fine-tuned instances of Buchberger's algorithm.

In addition to the seven components above, we have some more ambitious objectives that we attempt to partially answer within this thesis:

(8) Demonstrate the usability of software specialization as defined by the specifications framework defined in this thesis. This objective is more focused on the usability of these specializations when code generation is concerned. Some earlier work by J. Carette [14, 16] in the generation of Gaussian Elimination software motivated this objective to demonstrate the usefulness of such programming techniques for other computer algebra problems, and perhaps for other non-algebra-related computer programs in the future.

(9) Show that Buchberger's algorithm can be specialized and tailored in such a way as to resemble the algorithms of other fine-tuned and restricted domain algorithms that were made specifically for a limited class of computational algebra problems. Evidence of this objective would be the generation of an instance of Buchberger's algorithm for solving a linear system of polynomial equations that, not only textually resembles an instance of the Gaussian Elimination algorithm, but also matches its performance in terms of run time and space complexity.

(10) Show that it is possible to provide a proof of correctness for a specialization of a software product that we generated. This relies on an assumption that the generalized algorithm — being the full implementation of Buchberger's algorithm in the context of this thesis — is correct since we are not concerned with proving whether a certain computer program conforms to its specification or whether an algorithm is a solution to a given problem. However, we are concerned that, given a program which implements a specification for a problem, we can automatically deduce that a mechanically generated specialization of this program is an implementation of the specification for the specialized problem. We set up the formal framework for reasoning with specializations of software specifications and reasoning with the implementations of specifications, but providing the proof generator as a companion to the code generator is out of the scope of this thesis.

(11) Using this code generator for specialized instances of Buchberger's algorithm, we plan to produce some instances of the Gröbner bases solvers for a specific set of problems that have not been seen previously. This does not mean that we aim to produce genuinely new algorithms that were unknown to mankind before, but instead we are producing an algorithm for a problem with a very specific case of parameters and conditions by composing components from different implementations in such a way that this newly composed algorithm was not programmed before. This is especially useful for cases of problems where the current solution is simply to employ a full version of Buchberger's algorithm (or any of the derivative algorithms) without ever fine tuning the algorithm for this specific problem.

1.2 Organization

The organization of the chapters in this thesis follows closely the path set by the work plan as defined in §1.1.

Chapter 2 starts off with an overview of the mathematical background required to understand the theory of Gröbner bases and its applications in computational algebra. We also lay the foundations for the algebraic objects as they will be specified and programmed in later chapters. Finally, this chapter presents a version of Buchberger's algorithm that will be used directly as the foundation of our code generator.

Chapter 3 — independently of the previous chapter — describes the elements of meta programming in F# and defines a theoretical framework for reasoning about meta programs. Although we will not directly use this framework in the code generator or any of the proof techniques in this thesis, it defines the required theoretical background for the domain-specific language that we developed for this thesis. In addition to the meta programming framework and the code generation DSL, this chapter also introduces a new extension to the F# compiler and libraries that we programmed specifically for this research to accommodate nested quotations and splicing. This extension greatly enhances the abilities and the readability of multistage programs in F# and is necessary for the code generator presented here.

Chapter 4 outlines an approach to writing and reasoning with software specifications that we utilize in the course of this thesis. We provide a proper distinction between the multiple meanings of the word "abstraction" as commonly used in the

software literature and derive four concepts from this definition: Abstraction, refinement, conceptualization and actualization. Then we use the formal framework of writing software specifications to properly define each of these terms, and finally define specialization and generalization of software components that is consistent with the existing software engineering literature in order to use them to prove the correctness of specialized algorithms. Finally, we define what a "module" means in the context of software architecture and briefly discuss the methods for decomposing software architectures into the modules we defined.

The heart of this research, Chapter 5, uses the foundations set up by the last three chapters to create the main code generator. We first start by writing a formal specification for each algebraic object as defined in §2 as a module using the same language and module design defined in §4. Then, we implement each of these modules as an interface in F# and program a generator for each of these algebraic objects using the framework defined in §3. We perform the same tasks on defining the modules, interfaces and generators for the polynomial algebra for computational algebra. Next, we provide a modular decomposition of the version of Buchebrger's algorithm outlined in §2 and produce the module specifications, implementation and generators for each of these modules. Finally, we list the main code generator that uses all of these modules to generate specialized instances of Buchberger's algorithm and provide some samples of the usage and algorithms produced by this generator.

In Chapter 6 we demonstrate the most powerful aspect of the code generator from the previous sections to generate specialized algorithms for some sub-problems of Gröbner basis, such as those highlighted in §2.4. We will tailor the design and generation of these algorithms in such a way that they no longer resemble the full version of Buchberger's algorithm, but instead can clearly be seen as specialized instances of the sub algorithms defined. We choose the example of linear polynomial systems to generate a solver program that not only solves the system of equations in the same time-complexity class of Gaussian Elimination, but also resembles the algorithm for Gaussian Elimination textually and functionally. Chapter 6 also defines a simple specialization of Buchberger's algorithm for univariate polynomials to generate an instance of the Euclidean Algorithm.

In the conclusion, we will revisit the objectives outlined in §1.1 and recap how they were addressed throughout this thesis with references to the related sections. We also discuss the long-term objectives for this research and discuss what objectives were addressed by this research and what still remains at large.

1.3 Contributions

It is customary and important to individually list each contribution that we have made throughout the course of this research. We repeat here the parts of the thesis described above that we have contributed:

(1) A simple framework for reasoning about the syntax of F# using its meta programming elements (§3.1.1 and §3.1.2). The complete details of frameworks for reasoning about syntactic structures is in the process of publication as a series of papers by W.M. Farmer and P. Larjani [31].

(2) An extension to the splicing mechanism in F# compiler and libraries (§3.1.3) that allows variables defined within a quotation to be used within an inner spliced expression while ensuring that the variables do not escape their scope. This was a shortcoming in the F# language where such usage of variables was not compilable. This work was inspired by a similar feature in MetaOCaml by Walid Taha [89].

(3) A pretty-printer for F# quotations that prints out an expression in F# syntax instead of the abstract syntax tree printer that is provided by the F# libraries. This is a cosmetic feature that greatly improved the readability of the quotations produced by the code generator. All of the results presented in this thesis have been produced using this extension. Our pretty-printer is described in §3.1.3.

(4) A code generation domain-specific language is presented in §3.2 and is programmed as a very specific "computation expression'' in F#. In addition to this DSL, many code combinators are also provided to ease the task of writing code generators. All of the generators in this thesis are programmed using this language.

(5) The distinction and disambiguation between the multiple uses of the word "abstraction" in the standard software engineering literature is made in §4 where we distinguish between the ideas of abstraction and conceptualization — and their counterparts of refinement and actualization which are perhaps more important in usage — to present orthogonal definitions of the two concepts.

(6) A formal definition of software specialization within the context of specifications is defined in §4.1.2. We aim to make this definition in a consistent manner

with the current usage of the word "specialization" within most of the software engineering literature as well as to produce a logical definition with which we can prove correctness of specialized implementations of software specifications.

(7) The specifications for the algebraic modules within the context of computational algebra are defined in §5.1 using the same software specification language established in §4. This section also implements a library of F# interfaces and generators for these algebraic modules in addition to the module specifications, interfaces and implementations of polynomial algebra objects. This section establishes a rich algebra library for implementing — both in standard functions and in code generators — computational algebra programs in F#. This framework is similar to those used in other computer algebra and mechanized mathematics systems but has the advantage that we can directly program the algebraic algorithms within the F# programming language and take advantage of other .NET framework libraries without the need of an external system.

(8) A modular breakdown of Buchberger's algorithm is presented in §5.2 in a similar fashion of the modular decomposition methodology described by D.L. Parnas [65]. These modules are specified in the same language and framework defined earlier and the F# interfaces for these modules are provided. We also provide the generator for these modules and some generic and sample implementations which utilize the same algebra framework established earlier.

(9) The final code generator for producing specialized instances of Buchberger's algorithm is provided in §5.3. This generator is also programmed in the code generation DSL contributed by this thesis and generates the possible architectures corresponding to the modular breakdown in §5.2. The elements of this generator closely resemble the algorithms defined in §2. Several generated instances of Buchberger's algorithm are provided to demonstrate the extent of configuration and specialization provided by this generator with the focus on their resemblance to the classical algorithms for solving the corresponding problems.

Chapter 2

Computational Algebra

As mentioned in the introduction section, we are using Gröbner bases as a method of solving for ideal membership within systems of multivariate, non-linear, polynomial equations. This chapter is dedicated to the definition and explanation of Gröbner bases and the computational algorithms for finding them. No prior knowledge of Gröbner bases is required for the reader and this chapter contains the relevant information needed for the rest of the thesis. We also provide a brief overview of (commutative) abstract algebra and polynomial rings; however, we assume the reader is familiar with mathematical logic and basic algebra.

A Gröbner basis is a specific choice of basis elements for a polynomial ideal — which is not necessarily the smallest basis — that has a special property regarding polynomial division: The remainder of dividing any polynomial by these basis members is always unique, regardless of the order of the dividing elements. A precise explanation of this definition will be discussed later in this chapter and several competing, yet equivalent, definitions will be provided. The reader may also refer to Bruno Buchberger's original paper [11] for a full description. Weispfenning and Becker [96] and Kreuzer and Robbiano [48] also provide several algebraic approaches to defining Gröbner bases while Cox, Little and O'Shea [24] provide a geometric approach to presenting this definition.

We demonstrate the importance of having a unique remainder by the following example:

Example 2.1 Consider the polynomial ring $\mathbb{R}[x]$ and the pair of polynomials $\langle b_1, b_2 \rangle$ where $b_1 = x^2$ and $b_2 = x^2 + 1$. The goal is to determine if the polynomial $p = x^3 + x$ can be represented as a linear combination of b_1 and b_2. That is, for some polynomials

p_1 and p_2, we have $p = p_1 b_1 + p_2 b_2$. In other words, we would like to decide if the polynomial p is a member of the ideal spanned by $\langle b_1, b_2 \rangle$. An automated method may attempt to repeatedly divide this polynomial by the basis elements until the remainder of the division is no longer divisible by any of the elements and, if this remainder is 0, then we have a representation of p in this basis. For this example, dividing $x^3 + x$ by x^2 would lead to a divisor x and a remainder x. This remainder is no longer divisible by either b_1 or b_2, and thus we have the decomposition $p = xb_1 + 0b_2 + x$ which might suggest that p is not a member of the ideal $\langle b_1, b_2 \rangle$; however, it can be easily seen that $p = 0b_1 + xb_2$ (with no remainder) if we were to divide by b_2 first instead of b_1. This example demonstrates that some choices of the ordering for the basis elements may lead to incorrect decisions when testing for ideal membership using division. What makes this choice more difficult is that the "correct" choice of ordering differs from one polynomial to the other. For example, the polynomial $q = x^3$ can only be seen as a member of $\langle b_1, b_2 \rangle$ if we divide by b_1 first and then by b_2, completely the opposite of the order needed for p.
□

In general, if there are n basis elements for an ideal, one would have to make $n!$ attempts at division in the worst case to be certain that a polynomial is a member of this ideal or not — one attempt for every possible permutation of the basis elements. Clearly, this is not acceptable in terms of practicality and efficiency. A Gröbner basis for an ideal solves this issue (and many more!) by making the ordering of the basis elements *irrelevant* to the outcome of the algorithm.

This chapter is organized into four sections. The first section gives a brief overview of the algebraic preliminaries required for defining Gröbner bases. Readers who are well familiar with commutative algebra may only need to review §2.1.2 for some important terminology regarding quotients that will be used throughout this thesis. For more background information on algebraic concepts, readers may refer to Dummit and Foote [29] for abstract algebra and Kreuzer et al. [48, 49] for polynomial algebra definitions.

The second section of this chapter defines the computational aspects of polynomials over commutative rings and defines Gröbner bases and the problems that led to their discovery. This section also gives a brief overview of other problems that are either special (reduced) cases of Gröbner basis computation or can be encoded into one. Several of the references mentioned earlier [24, 48, 96] may also be used as

supplementary material for readers.

The third section of this chapter explains Buchberger's algorithm [11] for the computation of Gröbner bases and discusses the time/space complexity of this algorithm as well as several extensions and optimizations available for this computation.

Finally, we conclude this chapter by showing some of the important applications of Gröbner bases computation as either specializations of the problem or embeddings of other problems as finding Gröbner bases.

2.1 Polynomial Algebra

Before describing any of the computational challenges concerning polynomials, we will review some of the basic concepts in algebra and define the required theoretical background for working with polynomials. We emphasize that a "polynomial" throughout this thesis is defined as a general multivariate polynomial and univariate polynomials are the special case of polynomials in which only one variable is used. For an n-variable polynomial, we will use the notation $\vec{x}$ to mean the vector of variables $x_1, x_2, \ldots, x_n$. When no confusion may arise and $n \leq 3$, we may also refer to variables x_1, x_2, x_3 as x, y, and z, respectively.

2.1.1 Basic Algebra

We will start this section by reviewing the most basic structures that are used in this thesis. The notion of an *algebra* consists of a carrier set with some constants and operators. For example we may define the algebra $\mathcal{A} = (\Gamma, \star_\Gamma)$ as a carrier set Γ and a binary operator $\star_\Gamma : \Gamma \times \Gamma \to \Gamma$. For brevity, we may sometimes refer to the operators and constants by their unscripted names when there is no ambiguity.

Definition 2.2 (Commutative Monoid) A commutative *monoid* is a set Γ together with a commutative[1] associative[2] binary operation $\star : \Gamma \times \Gamma \to \Gamma$ and an element $\epsilon \in \Gamma$ such that $\forall_{\gamma \in \Gamma}\ \epsilon \star \gamma = \gamma \star \epsilon = \gamma$. We can represent this monoid as the algebra $\mathcal{M} = (\Gamma, \epsilon, \star)$.

Definition 2.3 (Group) A *group* is an algebra $\mathcal{G} = (\Gamma, \epsilon, \star, ^{-1})$ such that $(\Gamma, \epsilon, \star)$ is a monoid and $^{-1} : \Gamma \to \Gamma$ is a unary operator such that $\forall_{\gamma \in \Gamma}\ \gamma \star \gamma^{-1} = \gamma^{-1} \star \gamma = \epsilon$.

[1] *Commutativity* of an operator $\star : \Gamma \times \Gamma \to \Gamma$ is defined by the axiom $\forall_{\alpha,\beta \in \Gamma}\ \alpha \star \beta = \beta \star \alpha$.

[2] *Associativity* of an operator $\star : \Gamma \times \Gamma \to \Gamma$ is defined by the axiom $\forall_{\alpha,\beta,\gamma \in \Gamma}\ (\alpha \star \beta) \star \gamma = \alpha \star (\beta \star \gamma)$.

Definition 2.4 (Ring) A *ring* is an algebra $\mathcal{R} = (\Gamma, 0, 1, +, -, *)$ such that $(\Gamma, 0, +, -)$ is a group (called the *additive group*) and $(\Gamma, 1, *)$ is a commutative monoid (called the *multiplicative monoid*) such that the distributivity law $\forall_{\alpha,\beta,\gamma\in\Gamma}\ \alpha*(\beta+\gamma) = \alpha*\beta+\alpha*\gamma$ holds.

Readers familiar with ring theory may have already noticed that by "ring" we mean a "commutative ring with identity". This convention will be followed throughout this text.

Definition 2.5 (Field) A *field* $\mathcal{K} = (\Gamma, 0, 1, +, -, *, ^{-1})$ is a ring $(\Gamma, 0, 1, +, -, *)$ such that $(\Gamma\backslash\{0\}, 1, *, ^{-1})$ is a group[3] (called the *multiplicative group*).

When permitted, we may use other compound operators for brevity:

- Given any additive group, the binary operation $-$ is defined as $\alpha-\beta = \alpha+(-\beta)$.
- Given any field, the binary operation $/$ is defined as $\alpha/\beta = \alpha*(\beta^{-1})$ when $\beta \neq 0$ and undefined otherwise.
- Given any ring, the unary operation ^{n} when $n \in \mathbb{N}$ is defined as $\gamma^n = \gamma*\gamma*\cdots*\gamma$ (multiplication repeated n times — this is valid since $*$ is associative), with the following special cases[4,5]: $\gamma^0 = 1_\Gamma$ and $\gamma^{-n} = (\gamma^n)^{-1}$.

So far we have only defined some of the base structures in algebra. Before moving on to describing more structures, we first need to study two important meta-structures. These definitions do not directly define an algebra, but they first need to be applied to other algebras in order to produce the desired object.

Definition 2.6 (Homomorphism) A *homomorphism* of an algebra is a mapping which preserves all of the constants and operators of that algebra. For example, for any two rings $(\Gamma, 0_\Gamma, 1_\Gamma, +_\Gamma, -_\Gamma, *_\Gamma)$ and $(\Delta, 0_\Delta, 1_\Delta, +_\Delta, -_\Delta, *_\Delta)$, a *ring homomorphism* is a map $\varphi : \Gamma \to \Delta$ such that $\varphi(0_\Gamma) = 0_\Delta$, $\varphi(1_\Gamma) = 1_\Delta$ and $\forall_{\alpha,\beta\in\Gamma}\ \varphi(\alpha +_\Gamma \beta) = \varphi(\alpha) +_\Delta \varphi(\beta)$, etc.

[3] 0^{-1} is undefined.

[4] Note that the ring constant 1_Γ has been marked as being the multiplicative identity of $\mathcal{R}$ in order to avoid the confusion with the natural numbers used here.

[5] In general, 0_Γ^0 is undefined.

An *endomorphism* is a homomorphism of an object into itself, i.e., a homomorphism $\varphi : \Gamma \to \Gamma$ is called an endomorphism.

Definition 2.7 (Sub-algebra) Given a set Γ and some algebra $\mathcal{A}$ over this set, a subset $\Gamma' \subseteq \Gamma$ defines a *sub-algebra* $\mathcal{A}'$ of $\mathcal{A}$ if Γ' is closed under all the operations of $\mathcal{A}$ and contains all of the constants while all the axioms of $\mathcal{A}$ are necessarily satisfied by $\mathcal{A}'$. For example, given a group $\mathcal{G} = (\Gamma, \epsilon, \star, ^{-1})$ and a set $\Delta \subseteq \Gamma$ such that $\epsilon \in \Delta$, $\forall_{\alpha,\beta\in\Delta}\ \alpha \star \beta \in \Delta$ and $\forall_{\delta\in\Delta}\ \delta^{-1} \in \Delta$ then $\mathcal{H} = (\Delta, \epsilon, \star_\Delta, ^{-1}{}_\Delta)$ is a *sub-group* of $\mathcal{G}$ where $\star_\Delta, ^{-1}{}_\Delta$ are restrictions of the functions $\star, ^{-1}$ to the set Δ respectively. This relation of a sub-algebra $\mathcal{A}'$ to an algebra $\mathcal{A}$ is written as $\mathcal{A}' \leq \mathcal{A}$.

Notice that the image of an endomorphism always defines a natural sub-algebra. We now have the required background to define the composite algebras that we need in order to describe polynomial algebras. For the following definitions, assume $\mathcal{R} = (\Gamma, 0, 1, +, -, *)$ is a ring defined as above.

Definition 2.8 (R-MonoModule) An $\mathcal{R}$-*monomodule* $(\mathcal{M}, \cdot)$ is a monoid $\mathcal{M} = (\Delta, \epsilon, \star)$ together with an external ring of *coefficients* $\mathcal{R}$ and a binary operation $\cdot : \Gamma \times \Delta \to \Delta$ called the *scalar multiplication* such that the identity, associativity and distributivity laws are satisfied:

(1) **(Identity)** $\forall_{\delta\in\Delta}\ 1 \cdot \delta = \delta$.

(2) **(Associativity)** $\forall_{\alpha,\beta\in\Gamma,\delta\in\Delta}\ (\alpha * \beta) \cdot \delta = \alpha \cdot (\beta \cdot \delta)$.

(3) **(Additive Distributivity)** $\forall_{\alpha,\beta\in\Gamma,\delta\in\Delta}\ (\alpha + \beta) \cdot \delta = (\alpha \cdot \delta) \star (\beta \cdot \delta)$.

(4) **(Multiplicative Distributivity)** $\forall_{\gamma\in\Gamma,\alpha,\beta\in\Delta}\ \gamma \cdot (\alpha \star \beta) = (\gamma \cdot \alpha) \star (\gamma \cdot \beta)$.

Definition 2.9 (R-Module) An $\mathcal{R}$-*module* $(\mathcal{G}, \cdot)$ is a group $\mathcal{G} = (\Gamma, \epsilon, \star, ^{-1})$ with an external ring of coefficients $\mathcal{R}$ and scalar multiplication $\cdot : \Gamma \times \Delta \to \Delta$ which satisfies the same axioms as above. Alternatively, an $\mathcal{R}$-module $(\mathcal{G}, \cdot)$ is an $\mathcal{R}$-monomodule $(\mathcal{G}, \cdot)$ such that $\mathcal{G}$ is also a group.

Definition 2.10 (R-Algebra) An $\mathcal{R}$-*algebra* $(\mathcal{S}, \varphi)$ is a ring $\mathcal{S} = (\Gamma, 0, 1, +, -, *)$ with a ring homomorphism $\varphi : \Gamma \to \Delta$ of $\mathcal{R}$ called the *structural homomorphism*. Notice that if we define $\cdot$ as $\forall_{\gamma\in\Gamma,\delta\in\Delta}\ \gamma \cdot \delta = \varphi(\gamma) *_\Delta \delta$, then $(\mathcal{S}, \cdot)$ is an $\mathcal{R}$-module.

There are many other interesting definitions and theorems that arise from these structures which we cannot fully describe here. Readers may refer to the references mentioned earlier for more information. One very important sub-algebra of rings which will be the centre of computations in this thesis is the notion of an *ideal*.

Definition 2.11 (Ideal) An ideal I of a ring $\mathcal{R}$ is a subset $I \subseteq \Gamma$ which defines a subgroup of the additive group of $\mathcal{R}$ such that $\forall_{\gamma \in \Gamma, \delta \in I}\ \gamma * \delta \in I$ (or for short: $\Gamma * I \subseteq I$). For some $\gamma \in \Gamma$, we denote $\langle \gamma \rangle$ to be the smallest ideal $I \subseteq \Gamma$ such that $\gamma \in I$. Similarly, for $\Delta \subseteq \Gamma, \langle \Delta \rangle \subseteq \Gamma$ is the smallest ideal which contains all the elements of Δ.

In general, the ideal $\langle \Delta \rangle$ can be defined as the set of all linear combinations of elements of Γ with elements of Δ, i.e. for all elements $\delta_1, \cdots, \delta_n \in \Delta$ and $\gamma_1, \cdots, \gamma_n \in \Gamma$ we have that $\delta_1 * \gamma_1 + \cdots + \delta_n * \gamma_n$ is also in $\langle \Delta \rangle$. If for an ideal I we have that $I = \langle \Delta \rangle$ such that $|\Delta| < \infty$, then we say that the ideal I is *finitely generated* and the set Δ forms a *basis* for I. A ring $\mathcal{R}$ is said to be *finitely generated* if the trivial ideal $\langle 1_R \rangle = \mathcal{R}$ is finitely generated. We are only concerned with finitely generated rings and ideals in this thesis; thus we will always assume that by a ring $\mathcal{R}$ we mean a finitely generated commutative ring with identity.

2.1.2 Monomials and Terms

Fix $\mathcal{R} = (\Gamma, 0, 1, +, -, \cdot)$ to be a ring for the rest of this section. We are using the symbol $\cdot$ for ring multiplication from here on and reserving the symbol $*$ for monomial multiplication. Let $\vec{x} = x_1, \ldots, x_n$ be a vector of one or more variables in Γ. In this section we will explore how these variables form equations over the ring and the natural operations that arise from this definition.

Definition 2.12 (Monomial) A *monomial* m is a product $\prod_{i=1}^{n} {x_i}^{m_i} = {x_1}^{m_1} \cdot {x_2}^{m_2} \cdots {x_n}^{m_n}$ of variables where $m_1, \ldots, m_n \in \mathbb{N}$ and exponentiation in $\mathcal{R}$ is defined as earlier.

Monomials are the basic building blocks of equations using the variables in $\vec{x}$. We define the *degree* of a monomial to be the sum of exponents $\deg(\prod_{i=1}^{n} {x_i}^{m_i}) = \sum_{i=1}^{n} m_i$ with the usual addition on $\mathbb{N}$.

Let M be the set of all the monomials of $\mathcal{R}$ that can be made using $\vec{x}$, then there is an obvious isomorphism $\log : M \to \mathbb{N}^n$ such that $\log({x_1}^{m_1} \cdots {x_n}^{m_n}) = [m_1, \ldots, m_n]$ sometimes referred to as the *multidegree* of m.

Let $1_M \in M = \prod_{i=1}^{n} {x_i}^{0_\mathbb{N}}$, i.e., $\log(1_M) = [0_\mathbb{N}, \ldots, 0_\mathbb{N}]$ and $* : M \times M \to M$ be the binary operator such that $m * n = (\prod_{i=1}^{n} {x_i}^{m_i}) * (\prod_{i=1}^{n} {x_i}^{n_i}) = \prod_{i=1}^{n} {x_i}^{(m_i+n_i)}$, i.e. $\log(m * n) = \log(m) +_{\mathbb{N}^n} \log(n)$ where $+_{\mathbb{N}^n}$ is the normal pointwise vector addition on natural numbers. Then $\mathcal{M} = (M, 1_M, *)$ is a monoid called the *monomial monoid* of $\mathcal{R}$ over indeterminates $\vec{x}$.

We now need to explain the notions of divisors and multipliers between monomials:

Definition 2.13 Let $m, n \in M$ be monomials such that $m, n \neq 1_M$.

(1) m is a *multiple* of n and n is a *divisor* of m if $\exists_{d \in M}\ m = n * d$. Then n *divides* m, written as $n|m$, if m is a multiple of n.

(2) $g \in M \backslash \{0_M\}$ is the *greatest common divisor* of m and n if $g|m$ and $g|n$, and if any other non-zero monomial g' divides both m and n, then g' must also divide g. The greatest common divisor of m and n is unique and may be written as $\gcd(m, n)$ or $\gcd_{m,n}$.

(3) $l \in M$ is the *least common multiple* of m and n if $m|l$ and $n|l$ and l divides any other element l' that has this property. This may also be written as $\mathrm{lcm}(m, n)$ or $\mathrm{lcm}_{m,n}$.

(4) $\tau \in M$ is the *crossfactor* of m to n if $m * \tau = \mathrm{lcm}_{m,n}$. The crossfactor of m to n may be written as $\tau_{m,n}$.

Figure 2.1 demonstrates the relations between these concepts. Notice that gcd, lcm and τ always exist and in the extreme case scenario for m and n, $\gcd_{m,n} = 1_M$, $\mathrm{lcm}_{m,n} = m * n$ and $\tau_{m,n} = n$. In such a case we say that m and n are *relatively prime*.

A simple way of defining the greatest common divisor is $\gcd(\prod {x_i}^{m_i}, \prod {x_i}^{n_i}) = \prod {x_i}^{\min(m_i,n_i)}$ where min is the minimum function defined over $\mathbb{N}$. Similarly, we can define lcm using maximums of the exponents.

Example 2.14 Let $m = x_1 x_2^2 x_3^5$ and $n = x_2^3 x_3^3$ be two monomials in M. Then $m * n = x_1 \cdot x_2^2 \cdot x_3^5 \cdot x_2^3 \cdot x_3^3 = x_1 x_2^5 x_3^8$ is the product of the two monomials. Both m and n are divisible by $g = x_2^2 x_3^3$ which is the greatest common divisor $\gcd_{m,n}$ obtained by

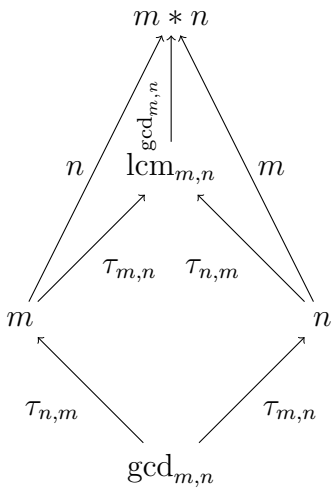

Figure 2.1: The relations between monomial divisors and multiples

taking the minimum exponent of each variable within the given monomials. We can easily verify that $g|m$ and $g|n$.

Similarly, by taking the maximum exponent of the m and n we obtain $l = x_1x_2^3x_3^5$ which is the least common multiple $\mathrm{lcm}_{m,n}$ of m and n. We can also verify easily that $m|l$, $n|l$, $l|m * n$, and that $g * l = m * n$.

Finally, we compute that $\tau_{m,n} = \mathrm{lcm}_{m,n}/m = n/\gcd_{m,n} = x_2$ and that $\tau_{n,m} = \mathrm{lcm}_{m,n}/n = m/\gcd_{m,n} = x_1x_3^2$. This completes all the relations shown in Figure 2.1. □

We now consider the monomials in M with coefficients in Γ denoted as $t = c \cdot m$ where $c \in \Gamma$ and $m \in M$. The choice of using $\cdot$ as the scalar multiplication here is not coincidental since after evaluation of the monomials with some variable assignment ($\vec{x} \leftarrow \vec{v} \in \Gamma^n$) this operation becomes the multiplication operator $\cdot$ of $\mathcal{R}$. The process of evaluation will be properly defined in §2.2. The $\cdot$ operation defines an $\mathcal{R}$-monomodule by the following definition:

Definition 2.15 (Term) A *term* of variables $\vec{x}$ in $\mathcal{R}$ is an element $t = c \cdot m$ where $c \in \Gamma$ is a constant (scalar) and $m \in M$ is a monomial with variables in $\vec{x}$. The induced $\mathcal{R}$-monomodule $\mathcal{T} = (\Gamma \times M, 1_T, *, \cdot)$ of these elements is called the *term monomodule* of $\mathcal{R}$ over the variables $\vec{x}$ where $1_T = 1_\Gamma \cdot 1_M$ and $*$ defined as $(c_1 \cdot m_1) * (c_2 \cdot m_2) = (c_1 \cdot_R c_2) \cdot (m_1 *_M m_2)$.

The notion of terms also entails some natural extensions of the ring and monomial operations. For example, the distributivity laws of $\cdot$ defines a partial summation $(c_1 \cdot m) + (c_2 \cdot m) = (c_1 +_R c_2) \cdot m$. Notice that this $+$ operator fails to define the addition of two terms whose corresponding monomials are not equal. We will later expand on these ideas to properly define addition and multiplication of polynomials.

When $\vec{x}$ has n variables, we may sometimes refer to the term monomodule of $\mathcal{R}$ over variables $\vec{x}$ as $\mathcal{T}^n$.

2.1.3 Term Orderings

Before we can fully explain the notion of polynomials here, we first need to define what ordering means between the elements of a term module. Assume $\mathcal{R}, \vec{x}, \mathcal{M}, \mathcal{T}$ are defined as earlier.

The purpose of ordering the elements of a set is to have a method of comparing individual items. In this chapter we are concerned about being able to compare terms and monomials (and later on, polynomials) to decide which one is "bigger". Ideally, we would like this definition of ordering to be consistent with the definition of least common multipliers and greatest common divisors that we defined earlier, so that the gcd is actually the biggest element (with respect to the ordering) in the set of all common divisors, and similarly have the lcm as the smallest (by the ordering) of the common multiples.

An *order* is a reflexive, transitive, antisymmetric relation on a set Γ. The following definition explains these terms and defines what a monomial ordering is.

Definition 2.16 (Monomial Ordering) Let $\mathcal{M} = (\Gamma, 1_\Gamma, *)$ be some monomial monoid of $\mathcal{R}$ over $\vec{x}$. A relation σ is a *monomial ordering* of $\mathcal{M}$, denoted by the inline comparison operator $\geq_\sigma$, if for all $\gamma_1, \gamma_2, \gamma_3 \in \Gamma$, we have:

(1) **(Least Element)** $\gamma_1 \geq_\sigma 1_\Gamma$.

(2) **(Reflexivity)** $\gamma_1 \geq_\sigma \gamma_1$.

(3) **(Antisymmetry)** $\gamma_1 \geq_\sigma \gamma_2 \wedge \gamma_2 \geq_\sigma \gamma_1 \Rightarrow \gamma_1 = \gamma_2$.

(4) **(Transitivity)** $\gamma_1 \geq_\sigma \gamma_2 \wedge \gamma_2 \geq_\sigma \gamma_3 \Rightarrow \gamma_1 \geq_\sigma \gamma_3$.

(5) **(Monotonicity)** $\gamma_1 \geq_\sigma \gamma_2 \Rightarrow \gamma_1 * \gamma_3 \geq_\sigma \gamma_2 * \gamma_3$.

We use other related notations to $\geq_\sigma$ for convenience, such as $\gamma_1 \leq_\sigma \gamma_2$ to mean $\gamma_2 \geq_\sigma \gamma_1$, $\gamma_1 >_\sigma \gamma_2$ to mean $\gamma_1 \geq_\sigma \gamma_2$, and $\gamma_1 \neq \gamma_2$, and also a similar definition for $<_\sigma$. For brevity, when a σ is defined in context, we may simply refer to these relations as $\geq$ etc.

Before continuing with examples of some orderings and their role in polynomials, we would like to know if the definition of ordering defined above is consistent with the meaning of the words "greatest" and "least" as defined for gcd and lcm.

Lemma 2.17 Let $\mathcal{M} = (\Gamma, 1, *)$ be a monomials monoid and $\geq$ a monomial ordering of $\mathcal{M}$. Then for any $m, n \in \Gamma$ such that $m|n$, we have $n \geq m$.

Proof Let $d \in \Gamma$ be the divisor such that $n = m * d$. We know such d exists by 2.13(1) because $m|n$.

$$\begin{array}{lll} & d \geq 1 & \text{by 2.16(1)} \\ \Rightarrow & d * m \geq 1 * m & \text{by 2.16(5)} \\ \Rightarrow & n \geq m & \end{array}$$

□

Theorem 2.18 Let $\mathcal{M} = (\Gamma, 1, *)$ be a monomial monoid and $\geq$ a monomial ordering of $\mathcal{M}$. Then for any $m, n, d \in \Gamma$ such that $d|m$ and $d|n$ we have $\gcd_{m,n} \geq d$. i.e. the gcd of two monomials is the largest of their common divisors with respect to $\geq$.

Proof Let $m, n \in \Gamma$ be two monomials and $d \in \Gamma$ such that $d|m$ and $d|n$. We know that $d|\gcd_{m,n}$ by 2.13(2). Lemma 2.17 proves $\gcd_{m,n} \geq d$. □

Corollary 2.19 Let $\mathcal{M}$ and $\geq$ be the same as in Theorem 2.18 and let $m, n, d \in \Gamma$ be such that $m|d$ and $n|d$, then $d \geq \text{lcm}_{m,n}$.

We can extend the definition of a monomial ordering $\geq_\sigma$ on $\mathcal{M}$ to a *term ordering* on $\mathcal{T}^n$ by the relation $(c_1 \cdot m_1) \geq_\sigma (c_2 \cdot m_2) \iff m_1 \geq_\sigma m_2$. Notice that this extension does not depend on any orderings that might exist on $\mathcal{R}$. We may now use $\geq_\sigma$ freely as both an ordering on $\mathcal{M}$ and $\mathcal{T}^n$ without ambiguity.

We will now define some of the common term orderings that will be used in this thesis. A desirable property for a term ordering is to be able to sort single-variable monomials by order of the variables, e.g. $x_1 \geq x_2 \geq x_3 \geq \cdots$. We would also like the monomials with different exponents of the same variable to be sorted in ascending order of their exponents, e.g. $x_1 \leq x_1^2 \leq x_1^3 \cdots$. The following definition expands these requirements into the multivariate case:

Definition 2.20 (Lexicographical term ordering) For any t_1, t_2 terms of $\mathcal{T}^n$, we define a *lexicographical term ordering* $\geq_{\texttt{Lex}}$ such that $t_1 \geq_{\texttt{Lex}} t_2$ if and only if either $t_1 = t_2$ or the first non-zero component of $\log(t_1) -_{\mathbb{N}^n} \log(t_2)$ is positive.

It is easy to confirm that this definition satisfies the requirements of term orderings and matches the two properties desired earlier. Another desirable property of a term ordering is to be *degree-compatible*, meaning that $t_1 \geq_\sigma t_2 \Rightarrow \deg(t_1) \geq_{\mathbb{N}} \deg(t_2)$. The following defines such an ordering:

Definition 2.21 (Degree-reverse-lexicographic term ordering) For any t_1, t_2 terms of $\mathcal{T}^n$, we define *degree-reverse-lexicographic term ordering* $\geq_{\texttt{DRL}}$ such that $t_1 \geq_{\texttt{DRL}} t_2$ if and only if $t_1 = t_2$ or $\deg(t_1) >_{\mathbb{N}} \deg(t_2)$ or, in the case that $\deg(t_1) = \deg(t_2)$ and $t_1 \neq t_2$, the last non-zero component of $\log(t_1) -_{\mathbb{N}^n} \log(t_2)$ is negative.

There are many other interesting term orderings and related results that will not be used in this thesis. For more information refer to Kreuzer et al. [48] Chapter 1.4.

2.1.4 Polynomials

We would like to generalize the notion of an equation with indeterminates within a ring. Earlier we explored how monomials can represent any product (using the $\cdot$ operation in ring) of variables and the concomitant concepts associated with them such as greatest common divisors. The concept of terms introduced the addition of scalar multiplication by ring elements to monomials, which added the support for scalars, constants (scalar multiplication by the monomial 1_Γ) and negations. The only ring operation missing from our definition of equations is the $+$ addition operator, which leads to the definition of polynomials. In this section we will define polynomials and show some of the basic operations and properties of polynomials.

Let $\mathcal{R} = (\Gamma, 0, 1, +, -, \cdot)$ be a ring, $\vec{x} = [x_1, \cdots, x_n]$ be a vector of n variables in $\mathcal{R}$ and $\mathcal{M} = (M, 1, *)$ be the monomial monoid of $\vec{x}$ variables over $\mathcal{R}$.

Definition 2.22 (Polynomial) $\rho : M \to \Gamma$ is a *polynomial* of n variables in $\mathcal{R}$ if only finitely many monomials assume a non-zero value by ρ. Alternatively, if only for $i = 1, \ldots, k$; $k < \infty$ we have $\rho : m_i \mapsto c_i$, $c_i \neq 0$, then we can view ρ as a set of terms $\{c_1 \cdot m_1, \ldots, c_k \cdot m_k\} \subset \mathcal{T}^n$ such that no two terms in the set share the same monomial. Polynomials are frequently written as a sum of terms: $\rho = c_1 \cdot m_1 + \cdots + c_k \cdot m_k$. The set $\text{supp}(\rho) = \{m_1, \ldots, m_k\}$ is called the *support* of ρ.

Similar to the choice of $\cdot$ for scalar multiplication, we chose the $+$ symbol for representation of the polynomials since it closely relates to the ring $+$ operation during evaluation. Evaluation of polynomials is defined in §2.2.

There is a natural transformation from a monomial to a term defined as $m \to 1_\Gamma \cdot m$ and another transformation from a term to a polynomial defined as $t \to \{t\}$. Thus any monomial or term could also be seen as a polynomial without ambiguity. The special cases of polynomials $\{\}$ and $\{1_T\}$ $(= \{1_\Gamma \cdot 1_M\})$ are represented as 0_ρ and 1_ρ respectively.

Define a $+$ operation between polynomials as the sum of their combined terms: $(c_1 \cdot m_1 + \cdots + c_k \cdot m_k) +_\rho (d_1 \cdot n_1 + \cdots + d_j \cdot n_j) = c_1 \cdot m_1 + \cdots + c_k \cdot m_k + d_1 \cdot n_1 + \cdots + d_j \cdot n_j$. This definition is not yet complete since it does not necessarily lead to a polynomial as result. For example, the sum of two polynomials $(c_1 \cdot m + \cdots) +_\rho (c_2 \cdot m + \cdots) = c_1 \cdot m + c_2 \cdot m + \cdots$ does not form a function $M \to \Gamma$ as the monomial m assumes two values by this sum. This problem is fixed by combining similar terms according to the partial rules of $+$ defined earlier between terms which define the distributivity over $\cdot$: $c_1 \cdot m + c_2 \cdot m = (c_1 + c_2) \cdot m$. Thus we define addition of polynomials as the sum of their combined terms after collecting similar terms. An immediate corollary to this definition is that for any polynomial ρ, $\rho +_\rho 0_\rho = \rho$.

Multiplication of polynomials is described as the sum of the pairwise term products: $\rho_1 \cdot_\rho \rho_2 = \sum_{t_1 \in \rho_1, t_2 \in \rho_2} t_1 *_T t_2$. Similarly, we define the negation $-_\rho (c_1 \cdot m_1 + \cdots + c_k \cdot m_k) = (-c_1) \cdot m_1 + \cdots + (-c_k) \cdot m_k$. Is it easy to confirm that all the constants and operators defined here form a ring.

Definition 2.23 (Polynomial Ring) Let P be the set of all polynomials with indeterminates $\vec{x}$ in $\mathcal{R}$, then $\mathcal{R}[\vec{x}] = (P, 0_\rho, 1_\rho, +_\rho, -_\rho, \cdot_\rho)$ is a ring of polynomials called the *polynomial ring* of $\mathcal{R}$ over variables $\vec{x}$.

Let φ be a ring homomorphism of $\mathcal{R}$ to $\mathcal{R}[\vec{x}]$ that defines the natural embedding of constants as polynomials: $\varphi : \gamma \mapsto \gamma \cdot 1_M$. The homomorphism φ defines $\mathcal{R}[\vec{x}]$ as an $\mathcal{R}$-Algebra and the scalar multiplication $\gamma \cdot \rho = (\gamma \cdot 1_M) \cdot \rho = (\gamma \cdot c_1) \cdot m_1 + \cdots + (\gamma \cdot c_k) \cdot m_k$. We will call $\mathcal{R}[\vec{x}]$ the *polynomial algebra* of $\mathcal{R}$ over $\vec{x}$ variables for the rest of this thesis, and the homomorphism above shows that we can use this $\mathcal{R}$-algebra to refer both the coefficient ring $\mathcal{R}$ and the polynomial ring $\mathcal{R}[\vec{x}]$ without ambiguity. We may also refer to the carrier set of the polynomial algebra simply by the name $\mathcal{R}[\vec{x}]$ when no confusion arises.

The notion of a term ordering induces a sorted structure on the terms within a polynomial such that for any given polynomial ρ and a term ordering $\geq_\sigma$, we can always find an enumeration of the terms within this polynomial such that $\rho = t_1 + t_2 + \cdots + t_k$ and $t_1 >_\sigma t_2 >_\sigma \cdots >_\sigma t_k$ [6]. The sorting of the terms within a polynomial produces a canonical representation of polynomials written in decreasing order of the terms.

Definition 2.24 (Leading Term) Let $\rho \in \mathcal{R}[\vec{x}]$ be a polynomial and σ a term ordering on $\mathcal{R}[\vec{x}]$. The *leading term* of ρ is a term $\mathrm{LT}_\sigma\rho \in \rho$ such that $\forall_{t\in\rho}\ \mathrm{LT}_\sigma\rho \geq_\sigma t$, i.e., t is the first term that appears in the canonical representation of ρ with respect to σ. If $\rho = 0_\rho$ then we set $\mathrm{LT}_\sigma\rho = 0 \cdot 1_M$. When $\mathrm{LT}_\sigma\rho = c \cdot m$, then $\mathrm{LC}_\sigma\rho = c$ is the *leading coefficient* of ρ and $\mathrm{LM}_\sigma\rho = m$ is the *leading monomial* of ρ. A polynomial ρ is called *monic* if $\mathrm{LC}_\sigma\rho = 1$.

Finally, we will expand the definition of $\geq_\sigma$ for polynomials and define the polynomial ordering: $\rho_1 \geq_\sigma \rho_2 \iff \mathrm{LT}_\sigma\rho_1 \geq_\sigma \mathrm{LT}_\sigma\rho_2$ (i.e. $\mathrm{LM}_\sigma\rho_1 \geq_\sigma \mathrm{LM}_\sigma\rho_2$) and the degree of a polynomial: $\deg_\sigma(\rho) = \deg(\mathrm{LM}_\sigma\rho)$. When σ is degree-compatible, then we additionally have $\deg_\sigma(\rho) = \max_{\mu\in\mathrm{supp}(\rho)}\ \deg(\mu)$.

2.2 Computation on Polynomials

We mentioned that the computational aspects of polynomials is a key topic in this thesis. A common constraint imposed on computational problems is that every data representation and execution must be finite due to the limitations of computer systems. This means that we will always be working with finitely generated rings, and more generally, finitely generated polynomial algebras. In this section we will look at a key computational aspect in algebra, the evaluation of polynomials. Although the question of what a polynomial evaluates to is not a complex problem, the converse question is more intriguing: For what variable assignments does a polynomial evaluate to a certain value? More generally, for which variable assignments do two or more polynomials attain the same value? We will explain this question and its ramifications in this section.

[6] The equality case of $\geq_\sigma$ is excluded since if $i \neq j$, $c_i \cdot m_i \geq_\sigma c_j \cdot m_j$ and $c_j \cdot m_j \geq_\sigma c_i \cdot m_i$ then $m_i = m_j$ by definition of term ordering and thus ρ would not be a polynomial.

2.2.1 Evaluation of Polynomials

Earlier during the definition of polynomials we hinted that the polynomial operators behave similarly to the base ring operations under evaluation. Recall that the definition of homomorphism required the transformation between two algebras to preserve all the operators of their respective structures. We will utilize the earlier hints to define evaluation as an $\mathcal{R}$-algebra homomorphism.

Let $\mathcal{R}[\vec{x}]$ be the polynomial algebra defined in §2.1, $\mathcal{T}^n$ be the term monomodule, $\mathcal{M}$ be the monomial monoid, and $\mathcal{S}$ be an $\mathcal{R}$-algebra.

Definition 2.25 (Variable Assignment) Let S be the underlying set of elements for the $\mathcal{R}$-algebra $\mathcal{S}$. A *variable assignment* is a map $\psi : \vec{x} \to S$ which assigns a value in set S for each indeterminate in $\vec{x}$, i.e. $\psi : x_i \mapsto s_i$ for $i = 1 \dots n$.

A variable assignment ψ may simply be written as a vector of n values $\psi = \vec{v} \in \mathcal{S}^n$ when $\psi : x_i \mapsto v_i$ is the assignment.

Definition 2.26 (Polynomial Evaluation) Let $\varphi : \mathcal{R} \to \mathcal{S}$ be the structural homomorphism of $\mathcal{S}$ and ψ be a variable assignment of $\vec{x}$ in $\mathcal{S}$. For any $\rho \in \mathcal{R}[\vec{x}]$, $t_1, t_2 \in \mathcal{T}^n$, $m \in \mathcal{M}$, $x_i \in \vec{x}$, $\gamma \in \mathcal{R}$, the *evaluation* of a polynomial ρ with respect to φ and ψ, $\mathbf{E}^{\psi}_{\varphi}(\rho)$, is the $\mathcal{R}$-algebra homomorphism $\mathcal{R}[\vec{x}] \to \mathcal{S}$ inductively defined using the following rules:

- $\mathbf{E}^{\psi}_{\varphi}(x_i) = \psi(x_i)$
- $\mathbf{E}^{\psi}_{\varphi}(\gamma \cdot_T m) = \varphi(\gamma) \cdot_S \mathbf{E}^{\psi}_{\varphi}(m)$
- $\mathbf{E}^{\psi}_{\varphi}(t_1 *_T t_2) = \mathbf{E}^{\psi}_{\varphi}(t_1) \cdot_S \mathbf{E}^{\psi}_{\varphi}(t_2)$
- $\mathbf{E}^{\psi}_{\varphi}(t_1 +_T t_2) = \mathbf{E}^{\psi}_{\varphi}(t_1) +_S \mathbf{E}^{\psi}_{\varphi}(t_2)$

This also includes the axioms of φ as a ring homomorphism and $\mathbf{E}^{\psi}_{\varphi}$ as an $\mathcal{R}$-algebra homomorphism.

When φ is the identity endomorphism of $\mathcal{R}$, we may write the evaluation of polynomials as $\mathbf{E}^{\psi}$ or simply $\mathbf{E}$ if the variable assignment ψ is unambiguous from the context. If ψ is the variable assignment $\vec{x} \mapsto \vec{v}$ then we may write $\rho(\vec{v}) = \mathbf{E}^{\psi}(\rho)$ for brevity.

2.2.2 Solving Systems of Equations

Evaluation of polynomials is a simple computational task, but a natural question that arises from this definition is: For which variable assignments does a polynomial evaluate to a given value? Or equivalently: When is the value of a polynomial zero? We will have a brief look at such equations and simultaneous solutions to systems of equations.

Definition 2.27 (Polynomial Equations) Let φ be the structural homomorphism of the $\mathcal{R}$-algebra $\mathcal{S}$, $\rho \in \mathcal{R}[\vec{x}]$ be a polynomial and $\gamma \in \mathcal{S}$ a constant. An *equation* is a formula of type $\rho = \gamma$ and the *solution* to this equation is the set of variable assignments $\{\psi \mid \mathbf{E}^{\psi}_{\varphi}(\rho) = \gamma\}$.

When no specific value is provided, a polynomial ρ implicitly defines an equation $\rho = 0$ with φ being the identity homomorphism, and its solution set is the kernel of the homomorphism $\mathbf{E}_{\varphi}(\rho) : \mathcal{R}^n \to \mathcal{R}$. For the remainder of this thesis we will assume that an equation ρ always means this equation. We do not lose any generality since an equation $\rho = \gamma$ may always be written in the form $(\rho - \gamma) = 0$. The solutions to the polynomial equation $\rho = 0$ are also called the *roots* of ρ.

Example 2.28 $\mathbb{R}[x, y]$ is the polynomial algebra with variables x and y and coefficients in $\mathbb{R}$. Let $p = x^2 - y$ be both a polynomial and an equation of this algebra. The set of solutions to p is the set $\{x \mapsto t; y \mapsto t^2 | t \in \mathbb{R}\}$. For brevity, this may also be expressed as $\{(t, t^2) \mid t \in \mathbb{R}\}$. Notice that the solutions corresponding to x and y can be expressed as polynomials in $\mathbb{R}[t]$. More importantly, the solution to y can be expressed as a polynomial $p_y = x^2$ in $\mathbb{R}[x]$. Thus, we have effectively eliminated the variable y from the solutions.
□

The above example demonstrates that we may express solutions to an equation as polynomials in the same polynomial algebra. We can define an equivalence relation between polynomials such that the polynomials within the same class share the same set of solutions. It is obvious to see that if $\rho_1 = \rho_2$ (i.e. they assign the same coefficient for every monomial) then they have the same solutions, but the converse question is more interesting: If two polynomials have the same set of solutions, are they equal? We will attempt to answer this question in a later section. First, we will examine a set of equations and the intersection of their solutions:

Definition 2.29 (Systems of Equations) Let $P = \{\rho_1, \ldots, \rho_k\} \subseteq \mathcal{R}[\vec{x}]$ be a set of k polynomials in $\mathcal{R}[\vec{x}]$. The resulting *system of equations* is the set of equations $\rho_1 = 0; \ldots; \rho_k = 0$ and the solution to this system of equation is the assignment $\vec{x} = \vec{r}$ where $\vec{r}$ is the set of their common roots: $\{\psi \mid \mathbf{E}^{\psi}_{\varphi}(\rho_1) = 0 \wedge \cdots \wedge \mathbf{E}^{\psi}_{\varphi}(\rho_k) = 0\}$.

In general, finding the solution to a system of equations is a difficult problem. Later we will see how a Gröbner basis leads to a solution for a system of equations, and Buchberger's algorithm provides a computational method of computing a Gröbner basis. There are other algorithms for solving special cases of systems of equations, for example, given a system of linear equations[7] we may use the Gaussian Elimination technique to find solutions.

We now consider some of the problems that can be expressed as solutions of system of equations.

Theorem 2.30 Let $\rho_1, \cdots, \rho_k \in \mathcal{R}[\vec{x}]$ be polynomials, $\mathcal{I} \subseteq \mathcal{R}[\vec{x}]$ an ideal such that $\mathcal{I} = \langle \rho_1, \cdots, \rho_k \rangle$, and let $\Gamma \subseteq \mathcal{R}^n$ be the set of solutions to the system of equations $\{\rho_1; \cdots; \rho_k\}$. Then for any $\rho \in \mathcal{I}$ and $\vec{x} = \gamma \in \Gamma$, γ is a root of ρ.

Proof Let $\rho \in \mathcal{I}$ and $\gamma \in \Gamma$, we know that there exists polynomials $\varrho_1, \cdots, \varrho_k$ such that $\rho = \sum \varrho_i \cdot \rho_i$ for $i = 1 \cdots n$ by the definition of ideal. Then:

$$\begin{aligned}
\rho(\gamma) &= \mathbf{E}^{\gamma}\rho \\
&= \mathbf{E}^{\gamma}(\textstyle\sum \varrho_i \cdot \rho_i) \\
&= \textstyle\sum \mathbf{E}^{\gamma}(\varrho_i \cdot \rho_i) \\
&= \textstyle\sum (\mathbf{E}^{\gamma}\varrho_i \cdot \mathbf{E}^{\gamma}\rho_i) \\
&= \textstyle\sum (\mathbf{E}^{\gamma}\varrho_i \cdot 0) \\
&= 0
\end{aligned}$$

□

Thus finding a solution for an ideal basis provides a solution for any member of that ideal.

Definition 2.31 (Affine Varieties) Let $\rho_1, \cdots, \rho_k \in \mathcal{R}[\vec{x}]$ be a set of polynomials, the *affine variety* $\mathbf{V}(\rho_1, \cdots, \rho_k) \subseteq \mathcal{R}^n$ is the set of all points $\gamma \in \mathcal{R}^n$ such that $\rho_1(\gamma) = \cdots = \rho_k(\gamma) = 0$.

It is easy to see that the affine variety generated by a set of polynomials is the same as the set of solutions to the system of equations they produce. Affine varieties

[7] Polynomial ρ is linear if $\deg(\rho) \leq 1$.

provide a method of algebraically describing geometric shapes. For example, in $\mathbb{R}[x, y]$ the variety $\mathbf{V}(x^2 + y^2 - 1)$ describes a circle in $\mathbb{R}^2$ of radius 1 at the origin.

Let I be an ideal of $\mathcal{R}[\vec{x}]$, then $\mathbf{V}(I)$ is the set of points $\gamma \in \mathcal{R}^n$ such that $\forall_{\rho \in I}\ \rho(\gamma) = 0$. Theorem 2.30 shows that for any $P \subset \mathcal{R}[\vec{x}]^k$ such that P is a basis for the ideal I (i.e. $I = \langle P \rangle$), then $\mathbf{V}(I) = \mathbf{V}(P)$.

Conversely, let $V = \mathbf{V}(\rho_1, \cdots, \rho_k)$ be some affine variety, then the set $\mathcal{I}(V)$ generated by this variety is the set of polynomials $\rho \in \mathcal{R}[\vec{x}]$ such that $\forall_{\gamma \in V}\ \rho(\gamma) = 0$. It is easy to confirm that $\mathcal{I}(V)$ forms an ideal.

We may now attempt to answer a question posed earlier: If two polynomials share the same set of solutions, are they the same polynomial? More generally, if two systems of equations have the same set of common solutions, do they span the same ideal? We may pose the question as $\mathcal{I}(\mathbf{V}(I)) \stackrel{?}{\supseteq} I$.

Definition 2.32 (Radical Ideals) Let I be an ideal of $\mathcal{R}[\vec{x}]$, then the *radical ideal* $\sqrt{I}$ is the set of all polynomials $\rho \in \mathcal{R}[\vec{x}]$ such that $\exists_{i \in \mathbb{N}^{>0}}\ \rho^i \in I$. Note that $\sqrt{I} \supseteq I$ trivially.

Theorem 2.33 (Nullstellensatz) Let $\mathcal{K}$ be an algebraically closed field and I an ideal of $\mathcal{K}[\vec{x}]$. Then $\mathcal{I}(\mathbf{V}(I)) = \sqrt{I}$.

This theorem is due to David Hilbert [24] and answers the question posed earlier for algebraically closed fields. These are all important computational questions posed as solving systems of equations.

2.2.3 Ideal Membership

Let us revisit the problem in Example 2.1 and what we learned from it. Given the pair of polynomials $\rho_1 = x_1^2; \rho_2 = x_1^2 + 1$, we would like to know if any given polynomial $\rho \in \mathcal{K}[\vec{x}]$ is a linear combination of them. In other words, we would like to know if a given polynomial is a member of an ideal: $\rho \stackrel{?}{\in} \langle \rho_1, \rho_2 \rangle$. If we set up a system of equations from ρ_1 and ρ_2 we quickly realize that there are no solutions to this system, i.e., $\mathbf{V}(\rho_1, \rho_2) = \emptyset$ and $\langle \rho_1, \rho_2 \rangle = \mathcal{R}[\vec{x}]$, thus any polynomial is a member of this ideal and can be represented as a linear combination of these elements. Two questions arise from this result: Firstly, if we know $\rho \in I$, how can we find out a representation of ρ as a linear combination of basis elements I? Secondly, if the answer to the system of equations was not as simple as above, how can we check if ρ is a member of I?

We mentioned in Example 2.1 that we use polynomial division for finding out the membership in this ideal, dependent on the ordering. In this section we will survey how polynomial quotients work.

Definition 2.34 (Quotient Ring) Let I be an ideal of $\mathcal{R}$ and $\mathcal{S}$ an $\mathcal{R}$-algebra with structural homomorphism $\varphi : \mathcal{R} \to \mathcal{S}$ such that $\ker \varphi = I$, i.e.:

$$I = \{\gamma \in R \mid \varphi(\gamma) = 0_S\}$$

Then the image of φ forms a subring of $\mathcal{S}$ with members $\varphi(\gamma)$ represented as $\gamma + I$ for any $\gamma \in \mathcal{R}$. We also define the operations $(\gamma_1 + I) +_S (\gamma_2 + I) = (\gamma_1 +_R \gamma_2) + I$ and $(\gamma_1 + I) \cdot_S (\gamma_2 + I) = (\gamma_1 \cdot_R \gamma_2) + I$. This ring is called the *quotient ring* $\mathcal{R}/I$.

We may now pose the question differently: Given a polynomial ρ, an ideal $I \subset \mathcal{R}[\vec{x}]$ and the quotient ring with the homomorphism $\varphi : \mathcal{R}[\vec{x}] \to \mathcal{R}[\vec{x}]/I$, is $\varphi(\rho) \stackrel{?}{\in} 0 + I$? In order to answer this question, we first need to define polynomial division.

Definition 2.35 (Polynomial Division) Let $p, q \in \mathcal{R}[\vec{x}]$ be two polynomials such that $q \neq 0$, σ a term ordering, then we can represent p with two other polynomials d, r such that $p = d \cdot q + r$ and $\mathrm{LT}_\sigma r \not| \mathrm{LT}_\sigma q$ and $r <_\sigma q$. In this case, d is called the *divisor* of p/q and r is the *remainder* of p/q.

When the ring $\mathcal{R}$ is a Euclidean Domain, then the divisor and remainder in $\mathcal{R}[\vec{x}]$ both exist and are unique [48] and can be obtained using the following algorithm:

Algorithm 2.36 (Division Algorithm)

// Inputs $p, q \in \mathcal{R}[\vec{x}], q \neq 0$, returns (d, r) such that $p = d \cdot q + r$ and $\mathrm{LT}_\sigma r \not| \mathrm{LT}_\sigma q$.
let rec Divide(p, q) =
    if $\mathrm{LT}_\sigma p \not| \mathrm{LT}_\sigma q$ **then**
        return $0, p$
    else
        let $d' = (\mathrm{LC}_\sigma p / \mathrm{LC}_\sigma q) \cdot (\mathrm{LM}_\sigma p / \mathrm{LM}_\sigma q)$ // i.e. $\mathrm{LT}_\sigma p / \mathrm{LT}_\sigma q$
        let $p' = p - d' \cdot q$
        let $d, r = \mathrm{Divide}(p', q)$
        return $(d' + d, r)$

Polynomial division provides a natural homomorphism $\varphi : \mathcal{R}[\vec{x}] \to \mathcal{R}[\vec{x}]/\langle q \rangle$ such that if r is the remainder of the division p/q, then $\varphi(p) = r + \langle q \rangle$. Similarly, define

$I = \langle \rho_1, \cdots, \rho_k \rangle \subseteq \mathcal{R}[\vec{x}]$ and $\rho \in \mathcal{R}[\vec{x}]$, let r_1 = the remainder of ρ/ρ_1, r_2 = the remainder of r_1/ρ_2, ..., r_k = remainder of r_{k-1}/ρ_k, then $\varphi : \mathcal{R}[\vec{x}] \to \mathcal{R}[\vec{x}]/I$ can be defined as $\varphi(\rho_k) = r_k + I$. Recall from Example 2.1 that r_k might be a different representative of the same element in quotient ring for different orderings of $\rho_1 \dots \rho_k$, but if for any ordering, $r_k = 0$ then this division leads to a representation of ρ as a linear combination of $\rho_1 \dots \rho_k$ and proves $\rho \in I$. We would like to find a basis for I such that this division would lead to a unique representative for ρ regardless of the ordering.

2.2.4 Polynomial Rewrite Systems

We will now investigate the polynomial division question from the perspective of term rewriting. We will assume that the reader has some familiarity with term rewriting, but will briefly define the required terminology for polynomial rewriting in this section. Refer to J. W. Klop [46] for a comprehensive introduction to term rewriting systems.

We will assume that $\mathcal{R}$ has a field structure when concerned with polynomial rewrite systems.

Definition 2.37 (Polynomial Rewrite Rule) A *polynomial rewrite rule* is a special rewrite rule of the form $r : m \mapsto p$ such that m is a monomial and p is a polynomial. Unlike generic term rewriting, a polynomial rewrite rule respects the associativity and commutativity of the polynomial algebra when rewriting the terms; thus given a polynomial $\rho \in \mathcal{R}[\vec{x}]$, any term t of ρ that is divisible by monomial m can be rewritten using the following rule:

$$\forall_{\rho \in R[\vec{x}]} \ \forall_{t \in \mathrm{supp}(\rho)} \ m|t \Rightarrow t \mapsto (t/m) \cdot p$$

i.e., every occurrence of the monomial m in any term is replaced by the polynomial multiplication of that term (as a polynomial) by p.

Given a polynomial $f \in \mathcal{R}[\vec{x}]$ and a term ordering σ, we define the rewrite rule $\xrightarrow{f}$ to be the polynomial rewrite rule $\mathrm{LM}_\sigma f \ \mapsto \ -(\mathrm{LC}_\sigma^{-1} f) \cdot (f - \mathrm{LT}_\sigma f)$. That is, if $f = c \cdot m + f'$ with $\mathrm{LT}_\sigma f = c \cdot m$ then $\xrightarrow{f}: m \mapsto -c^{-1} \cdot f'$. Let $F = \{f_1, \dots, f_m\} \subset \mathcal{R}[\vec{x}]$ be a set of polynomials, then $\overset{F}{\twoheadrightarrow}$ is the reflexive transitive closure of the set of polynomial rewrite rules $\{\xrightarrow{f_1}, \dots, \xrightarrow{f_m}\}$.

Definition 2.38 (Reduction) Given the polynomials $f, \rho \in \mathcal{R}[\vec{x}]$, if there exists a polynomial ρ' such that $\rho \xrightarrow{f} \rho'$ then we say that ρ *reduces* to ρ' using rewrite rule f and $\rho \xrightarrow{f} \rho'$ is the *reduction step*. Similarly, if $F \subset \mathcal{R}[\vec{x}]$, $\rho \in \mathcal{R}[\vec{x}]$, $\rho' \in \mathcal{R}[\vec{x}] \backslash \{\rho\}$ and $\rho \overset{F}{\twoheadrightarrow} \rho'$, then ρ reduces to ρ' using F. If no such $\rho' \neq \rho$ exists such that $\rho \overset{F}{\twoheadrightarrow} \rho'$, then ρ is *irreducible* with respect to F.

There are two important results that arise from the definition of polynomial reductions: First is the relation of reduction to remainders of division, which links polynomial rewriting to the ideal membership problem defined earlier; second is the concept of confluence of a rewrite system which draws the link to the two questions posed earlier. We will explore these relations briefly.

Theorem 2.39 Let $p, f, r \in \mathcal{R}[\vec{x}]$ be polynomials and choose a term ordering on $\mathcal{R}[\vec{x}]$, then $p \xrightarrow{f} r$ if and only if r is the remainder of p/f.

Proof Let σ be a term ordering on $\mathcal{R}[\vec{x}]$ for this proof, and let $f = c \cdot m + f'$ such that $c = \mathrm{LC}_\sigma f$ and $m = \mathrm{LM}_\sigma f$ (i.e. $c \cdot m = \mathrm{LT}_\sigma f$).
$\Rightarrow$: Assume $p \xrightarrow{f} r$. Let $t \in \mathrm{supp}(p)$ such that $t | m$ and let $p = t + p'$, then $\xrightarrow{f}$ rewrites t to $(-c^{-1}) \cdot (t/m) \cdot f'$; therefore:

$$\begin{aligned} r &= (-c^{-1}) \cdot (t/m) \cdot f' + p' \\ &= (-c^{-1}) \cdot (t/m) \cdot f' + (p - t) && \text{since } p = t + p' \\ &= p - (c^{-1} \cdot t/m \cdot f' + t) \\ &= p - c^{-1} \cdot t/m \cdot f && \text{since } f = f' + c \cdot m \end{aligned}$$

Let $d = c^{-1} \cdot t/m$, then $p = d \cdot f + r$. We also know that $\mathrm{LT}_\sigma r \not| m$ since it was not rewritten by $\xrightarrow{f}$. Then $\mathrm{LT}_\sigma r \not| \mathrm{LT}_\sigma f$.
Therefore r is the remainder of the division p/f.
$\Leftarrow$: Assume r is the remainder of p/f, i.e. $\exists d \in \mathcal{R}[\vec{x}]$ such that $p = d \cdot f + r$ and $\mathrm{LT}_\sigma r \not| \mathrm{LT}_\sigma f$, then $p = d \cdot c \cdot m + d \cdot f' + r$. We know that no terms of f' divide m since $\mathrm{LT}_\sigma f >_\sigma f'$; therefore the term $d \cdot c \cdot m$ is the only part of p that is rewritten by $\xrightarrow{f}$:

$$\begin{aligned} p &= d \cdot c \cdot m + d \cdot f' + r \\ &\xrightarrow{f} (d \cdot c \cdot m/m) \cdot (-c^{-1} \cdot f') + d \cdot f' + r \\ &= (-d \cdot f') + d \cdot f' + r \\ &= r \end{aligned}$$

Therefore $p \xrightarrow{f} r$.
□

An immediate usage of this theorem is to show that every chain of rewrite rules is eventually stationary and ends in a "minimal" element. We can use the well-ordering principle on term orderings together with the following result to show that such minimal element exists and that the rewrite rule $\overset{F}{\twoheadrightarrow}$ is finite:

Corollary 2.40 $\rho \xrightarrow{f} \rho' \Rightarrow \rho \geq_\sigma \rho'$.

Proof Direct result of Lemma 2.17 and Theorem 2.39.
□

Given a set of polynomials $F = \{f_1, \ldots, f_m\} \subset \mathcal{R}[\vec{x}]$ and a polynomial $\rho \in \mathcal{R}[\vec{x}]$, we can conclude through our investigation in §2.2.3 and Theorem 2.39 that if $\rho \overset{F}{\twoheadrightarrow} 0$ then $\rho \in \langle F \rangle$, but we have already seen that the result of this reduction is highly dependent on the order which the polynomial rewrite rules $\xrightarrow{f_1}, \cdots, \xrightarrow{f_m}$ are applied. This brings us to the following definition:

Definition 2.41 (Confluent Rewrite System) A rewrite system $\overset{F}{\twoheadrightarrow}$ is called *confluent* if $\forall \rho_1, \rho_2, \rho_3 \in \mathcal{R}[\vec{x}]$ such that $\rho_1 \overset{F}{\twoheadrightarrow} \rho_2$ and $\rho_1 \overset{F}{\twoheadrightarrow} \rho_3$, then $\exists \rho_4 \in \mathcal{R}[\vec{x}]$ such that $\rho_2 \overset{F}{\twoheadrightarrow} \rho_4$ and $\rho_3 \overset{F}{\twoheadrightarrow} \rho_4$.

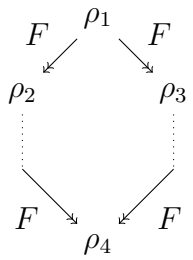

Figure 2.2: A confluent rewrite system.

Given a confluent and terminating polynomial rewrite system, we can reliably solve the ideal membership problem — and as we will see later, solve the systems of equations through elimination — by checking if a polynomial ρ rewrites to 0 using the rewrite system. This brings us two fundamental questions:

(1) Given a basis $\{f_1, \ldots, f_m\}$ for an ideal $I \subseteq \mathcal{R}[\vec{x}]$, how do we algebraically define an alternative basis $\{g_1, \ldots, g_k\}$ for I such that the resulting polynomial rewrite system $\overset{G}{\twoheadrightarrow}$ exists and is confluent?

(2) How do we compute such a basis?

The first question is answered by the definition of a Gröbner basis in §2.2.5. Although it is tempting (and possible!) to modify and use the Knuth-Bendix Completion algorithm [47] for computing the confluent *polynomial* rewrite system and re-interpret the result back as a set of polynomials, we have an algebraic method called Buchberger's Algorithm at our disposal that we will define in §2.3 to answer the second question [58][8].

2.2.5 Gröbner Bases

We have defined the problem of obtaining a "special" basis for an ideal such that the order of division does not change the final remainder when deciding the ideal membership question — alternatively, a basis for an ideal that leads to a confluent polynomial rewrite system. We will call this basis a Gröbner basis:

Definition 2.42 (Gröbner Basis) Let $\mathcal{R}[\vec{x}]$ be a polynomial ring, σ a term ordering on $\mathcal{R}[\vec{x}]$, $G = \{g_1, \ldots, g_k\} \subset \mathcal{R}[\vec{x}]$ and $I = \langle G \rangle$ be the ideal spanned by G. Then G is a *σ-Gröbner basis* for I if for any $\rho \in I$, there are $f_1, \cdots, f_k \in \mathcal{R}[\vec{x}]$ such that $\rho = \sum_{i=1}^{k} f_i \cdot g_i$ and $\mathrm{LT}_\sigma \rho \geq_\sigma \mathrm{LT}_\sigma(f_i \cdot g_i)$ for all $1 \leq i \leq n$.

Equivalently, we can say G is a Gröbner basis if $\overset{G}{\twoheadrightarrow}$ is confluent.

Before describing the uses for Gröbner bases and the methods for solving the questions posed earlier, we first need to ensure that every ideal in a polynomial ring has a Gröbner basis associated with it. Later we will define what is required to make such bases unique.

Theorem 2.43 (Hilbert Basis Theorem) If $\mathcal{R}$ is a finitely generated ring, then every ideal $I \subseteq \mathcal{R}[\vec{x}]$ has a finite basis.

Proof See [24] Theorem 4.
□

Knowing that there always exists a finite set of polynomials $f_1, \ldots, f_m$ such that $I = \langle f_1, \ldots, f_m \rangle$, we now only need to know that given a finitely generated ideal, there exists a Gröbner basis for it:

[8] In fact, the link between Buchberger's algorithm and Knuth-Bendix completion was unknown for years after Buchberger's discovery [12].

Theorem 2.44 (Existence of a σ-Gröbner Basis) Let $I \subseteq \mathcal{R}[\vec{x}]$ be a finitely generated ideal such that $I \neq \{0\}$ and σ be a term ordering on the terms of $\mathcal{R}[\vec{x}]$, then there exists $G = \{g_1, \ldots, g_k\}$ such that $I = \langle G \rangle$ and G is a σ-Gröbner basis.

Proof See [48] Theorem 2.4.3.
□

The existence of a Gröbner basis is an important result, but it does not mean that a σ-Gröbner basis of an ideal is unique. For example, both sets $G_1 = \{x\}$ and $G_2 = \{2x\}$ span the same ideal in $\mathbb{R}[x]$ and it is easy to check that G_1, G_2 are both Gröbner bases[9]. The set $G_3 = \{x, x^2\}$ is also a Gröbner basis for the same ideal. We may deduce that the bases G_2 and G_3 are not minimal because they contain "redundant" polynomials that are linear combinations of other basis members (such as $x^2 \in G_3$) or they may further be simplified by a constant factor (such as $2x \in G_2$).

Definition 2.45 (Minimal Gröbner Basis) A σ-Gröbner basis G for a polynomial ideal $I \subseteq \mathcal{R}[\vec{x}]$ is *minimal* if:

(1) Every basis member is *monic*: $\forall_{g_i \in G} \operatorname{LC}_\sigma g_i = 1_R$.

(2) For all $g_i \in G$, $\operatorname{LT}_\sigma g_i$ does not divide the leading term of any polynomial in $G \backslash \{g_i\}$.

A minimal Gröbner basis of I ensures that the size of the basis is minimal with respect to all the other Gröbner bases of I and that every basis element is monic. It is important to note that minimality of a Gröbner basis still does not entail its uniqueness. Let $G_4 = \{x, y\}$ and $G_5 = \{x + y, y\}$, then both G_4 and G_5 are minimal Gröbner bases with respect to $\geq_{\texttt{LEX}}$ term ordering (as per Definition 2.20). In fact, for any $g, h \in G$ such that $g \geq_\sigma h$ and G is a minimal Gröbner basis, we may replace g by $g' = g + a \cdot h$ for any constant $a \in R$ to obtain a new, equivalent, minimal Gröbner basis.

Definition 2.46 (Reduced Gröbner Basis) A σ-Gröbner basis G for a polynomial ideal $I \subseteq \mathcal{R}[\vec{x}]$ is *reduced* if:

(1) G is a minimal σ-Gröbner basis.

(2) For all polynomials $g_i \in G$ and terms $t \in \operatorname{supp}(g_i)$, t does not divide the leading term of any polynomial in $G \backslash \{g_i\}$.

[9] When there is no confusion, we may skip the term ordering σ on the Gröbner bases.

Theorem 2.47 (Uniqueness of Reduced σ-Gröbner Bases) Let $I \subseteq \mathcal{R}[\vec{x}]$ be a finitely generated ideal such that $I \neq \{0\}$ and σ be a term ordering on the terms of $\mathcal{R}[\vec{x}]$ and G be a reduced σ-Gröbner basis for I. Then G is unique.

Proof See [48] Theorem 2.4.13.
□

Therefore every polynomial ideal has a unique reduced Gröbner basis. This fact entails that every ideal $I \subseteq \mathcal{R}[\vec{x}]$ has a canonical representation of finitely many elements[10].

2.2.6 Elimination and Extension

Before discussing the method of computation for Gröbner bases, we may now review the questions posed earlier in this chapter and attempt to answer them. The ideal membership problem (second question) and the confluent polynomial rewrite system problem (third question) may now directly be solved using the reduced Gröbner basis for the corresponding ideals. The problem of solving systems of equations (first question), however, requires more explanation and draws on the important link between the ideals in algebra and varieties in geometry.

Theorems 2.30 and 2.33 showed that given an algebraically closed field $\mathcal{K}$, an ideal $I \subseteq \mathcal{K}[\vec{x}]$ such that I is a radical ideal, and any basis $\{\rho_1, \ldots, \rho_k\}$ spanning I, then the solutions to the system of equations $\{\rho_1 = 0; \cdots; \rho_k = 0\}$ defines the variety $\mathbf{V}(I)$. We will study the case of choosing the reduced Gröbner basis of I as the choice of basis for solving the system of equations. Although we will not discuss the case when I is not radical, the solutions to the bases of a non-radical ideal only differ from the variety by multiplicity of roots [76].

Definition 2.48 (Elimination Ideal) Let $I \subseteq \mathcal{R}[x_1, \ldots, x_n]$ be a polynomial ideal of n variables in $\mathcal{R}$, then the *i-th elimination ideal* of I is the ideal $I_i = I \cap \mathcal{R}[x_{i+1}, \ldots, x_n]$.

Given some ideal I, in the ideal I_i the first i variables in the polynomial field are eliminated and consequently any polynomial in I that uses the variables $x_1 \ldots x_i$ is eliminated from I_i. Thus the solutions to I_{n-1} (which only uses one variable x_n) can be solved using any known methods of finding the roots of a univariate polynomial.

[10] For the case of $I = \{0\}$, the empty set $\emptyset$ is the representation.

Then the results may be extended to solutions of I_{n-2} (using two variables x_{n-1} and x_n) by backwards substitution of occurrences of variable x_n to obtain new univariate polynomials of variable x_{n-1}. The same process of solving for one variable can be applied to find values of x_{n-1} and repeat the extension step until all solutions for the 0-th elimination ideal $I_0 = I$ are found. The following theorem shows that a basis for each elimination ideal can easily be found by using Gröbner bases:

Theorem 2.49 (The Elimination Theorem) Let $I \subseteq \mathcal{R}[x_1, \ldots, x_n]$ be a polynomial ideal of n variables in $\mathcal{R}$, σ be the lexicographical term ordering on terms of $\mathcal{R}[\vec{x}]$, and G be the reduced σ-Gröbner basis of I. Then for any $0 \leq i \leq n$, the set $G_i = G \cap \mathcal{R}[x_{i+1}, \ldots, x_n]$ is a Gröbner basis of the i-th elimination ideal I_i.

Proof See [24] Theorem 2.
□

Example 2.50 Consider the following system of equations in $\mathbb{C}[x, y, z]$:

$$\begin{cases} x^2 + y + z = 1 \\ x + y^2 + z = 1 \\ x + y + z = 1 \end{cases}$$

Let $f_1 = x^2 + y + z - 1; f_2 = x + y^2 + z - 1; f_3 = x + y + z - 1$ be the corresponding set of polynomials to this system and I be the ideal $\langle f_1, f_2, f_3 \rangle$. Using lexicographical term ordering, the reduced Gröbner basis for I is $G = \{g_1 = x + y + z - 1; g_2 = y^2 - y; g_3 = yz + \frac{1}{2}z^2 - \frac{1}{2}z; g_4 = z^3 - z\}$. §2.3.5 shows how this basis is computed.

The second elimination ideal I_2 — that is, eliminating variables x and y — has Gröbner basis $G_2 = \{g_4\}$ according to Theorem 2.49. To find the solutions to $g_4 = z^3 - z = 0$ we factor g_4 to $z(z-1)(z+1)$ and conclude that the possible values for z are $\{0; -1; 1\}$.

By substituting the possible values of z into the Gröbner basis for the first elimination ideal I_1, $G_1 = \{g_2; g_3; g_4\}$ we can obtain the possible values for y. First, substitute the choices $z = -1$ and $z = 1$ into $g_3 = yz + \frac{1}{2}z^2 - \frac{1}{2}z = 0$ to obtain the (partial) solution $(y, z) \in \{(1, -1); (0, 1)\}$, but for the remaining choice $z = 0$ we have infinitely many solutions $(y, z) \in \{(y, 0) \mid y \in \mathbb{C}\}$. Next, we substitute these solutions in equation $g_2 = y^2 - y = 0$ which only has solutions $y = \{0, 1\}$. Thus, the final set of solutions to the first elimination ideal is $(y, z) \in \{(1, -1); (0, 1); (0, 0); (1, 0)\}$.

Finally, we may substitute the solutions of I_1 into the 0th elimination ideal $I_0 = I$ for the full set of solutions. By substituting $(y, z) \in \{(0,0); (0,1); (1,0); (1,-1)\}$ in the equation $g_1 = x + y + z - 1 = 0$ we obtain the solutions $(x, y, z) \in \{(1,0,0); (0,0,1); (0,1,0); (1,1,-1)\}$ to the above system of equations.
□

2.3 Buchberger's Algorithm

Definition 2.42 for Gröbner bases is a non-constructive definition which does not inspire an immediate algorithm — unlike Definitions 2.45 and 2.46 which embody the methods for refining a Gröbner basis into the corresponding minimal and reduced Gröbner bases. This section defines Buchberger's Algorithm for computing Gröbner bases [12] and explores some improvements and optimizations to the original statement of the algorithm.

2.3.1 S-Polynomials and Normal Forms

Definition 2.51 (S-Polynomial) Let $\rho_1, \rho_2 \in \mathcal{R}[\vec{x}]$ be two polynomials and σ be a term ordering on terms of $\mathcal{R}[\vec{x}]$. Let $\tau_{\sigma,\rho_1,\rho_2} = \tau_{\mathrm{LM}_\sigma \rho_1, \mathrm{LM}_\sigma \rho_2}$ be the extension of the crossfactor τ from Definition 2.13 to polynomials, then the *S-polynomial* of ρ_1 and ρ_2 is defined as:

$$\mathbf{S}_{\sigma,\rho_1,\rho_2} = \tau_{\sigma,\rho_1,\rho_2} \cdot \rho_1 - \tau_{\sigma,\rho_2,\rho_1} \cdot \rho_2$$

When the choice of term ordering σ is unambiguous, we may simply refer to the S-Polynomial as $\mathbf{S}\rho_1, \rho_2$.

The S-polynomial of ρ_1 and ρ_2 multiplies both polynomials by their respective crossfactors[11] to homogenize the leading terms to be the least common multiple of both and then subtracts the two resultants to obtain a new polynomial with a different leading term. The following result is immediate from the definition of ideals and S-polynomials:

Proposition 2.52 Let $I \subseteq \mathcal{R}[\vec{x}]$ be a polynomial ideal, then $\rho_1, \rho_2 \in I \Rightarrow \mathbf{S}_{\rho_1,\rho_2} \in I$.

[11] Note that this term is made of both the crossfactor of the leading coefficients and the crossfactor of the leading monomials.

Definition 2.53 (Normal Remainder) Let $\{f_1, \ldots f_k\} \subset \mathcal{R}[\vec{x}]\backslash\{0\}$ be a set of polynomials for some $k \geq 1$ and $F = (f_1, \ldots, f_k)$ be an ordered k-tuple. Given a polynomial $\rho \in \mathcal{R}[\vec{x}]$ and a term ordering σ, let $d_1, \ldots, d_k, r \in \mathcal{R}[\vec{x}]$ be the divisors and the remainder such that $\rho = \sum_{i=1}^{k} f_i d_i + r$ where each d_i is obtained by applying the division algorithm with f_i to the remainder of the previous step r_{i-1} — we assume $r_0 = \rho$. Then the *normal remainder* of ρ with respect to the polynomials F and the ordering σ is defined as $\mathrm{NR}_{\sigma,F}(\rho) = r$. When σ can be determined by context, we may refer to the normal remainder as $\mathrm{NR}_F(\rho)$.

Computation of normal remainders is similar to that defined in §2.2.3 when the ordering of the quotients is predetermined by the ordering of the dividing polynomials and uses Algorithm 2.36. The following result is immediate from the definition of Gröbner bases:

Proposition 2.54 Let G be a σ-Gröbner basis for ideal I, then $\rho \in I \Leftrightarrow \mathrm{NR}_{\sigma,G}\rho = 0$.

Moreover, the two earlier propositions entail that $\rho_1, \rho_2 \in I \Rightarrow \mathrm{NR}_G \mathbf{S}_{\rho_1,\rho_2} = 0$.

2.3.2 Buchberger's Algorithm

Theorem 2.55 (Buchberger's Criterion) Let $G = \{g_1, \ldots, g_k\} \subset \mathcal{R}[\vec{x}]\backslash\{0\}$ be a set of non-zero polynomials, σ a term ordering on terms $\mathcal{T}^n$ and $I = \langle G \rangle$ be a polynomial ideal. Then G is a σ-Gröbner basis for I if and only if $\forall_{g_i,g_j \in G}\ \mathrm{NR}_{\sigma,G}\mathbf{S}_{\sigma,g_i,g_j} = 0$.

This is a constructive alternative to Definition 2.42 which can be implemented simply by computing the S-polynomials and adding the non-zero normal remainders to the basis until this condition is satisfied:

Algorithm 2.56 (Buchberger's Algorithm)
Inputs $F \subset \mathcal{R}[\vec{x}]\backslash\{0\}$, returns $G \subset \mathcal{R}[\vec{x}]$ such that $\langle G \rangle = \langle F \rangle$ and G is a Gröbner basis.

```
1 let Gröbner(F) =
2     let G = F
3     let B = {⟨i, j⟩ | 1 ≤ i < j ≤ |G|}
4     while B ≠ ∅ do
5         for ⟨i, j⟩ ∈ B do
6             B ← B − {⟨i, j⟩}
7             let s = S_{G_i,G_j}
```

```
8            let s' = NR_G s
9            if s' ≠ 0 then
10               G ← G ∪ {s'}
11               B ← B ∪ {⟨i, |G|⟩ | 1 ≤ i < |G|}
12       return G
```

The computation of normal remainders in Buchberger's algorithm can be done either by performing long division on polynomials or through polynomial rewriting. Theorem 2.39 showed that the remainder obtained using the two methods is always equal. We will analyse the termination and run-time complexity of this algorithm in §2.3.4.

2.3.3 Minimal and Reduced Gröbner Bases

Algorithm 2.57 (Minimal Gröbner Basis)
Inputs a Gröbner basis G, returns G_M such that $\langle G\rangle = \langle G_M\rangle$ and G_M is a minimal Gröbner basis.

```
let MinimalGröbner(G) =
    let G' = G and G_M = ∅
    for g ∈ G do
        G' ← G' − {g}
        if ∀_{g'∈G'} LT_σ g' ∤ LT_σ g then
            G_M ← G_M ∪ {monic_σ(g)}
    return G_M
```

Algorithm 2.58 (Reduced Gröbner Basis)
Inputs a polynomial p and basis G, reduces p with respect to G such that $\forall_{t\in\mathrm{supp}(p),g\in G}\ t \nmid \mathrm{LT}_\sigma g$.

```
let rec Reduce(p, G) =
    let p' = p
    for t ∈ supp(p) do
        for g ∈ G do
            if t | LT_σ g then
                p' ← p − (t/LT_σ g) · g
    if p ≠ p' then
        return Reduce(p', G)
    else
        return p
```

Inputs a minimal Gröbner basis G, returns the reduced Gröbner basis G_R such that $\langle G \rangle = \langle G_R \rangle$.

let ReducedGröbner(G) =
    let $G_R = \emptyset$
    for $g \in G$ **do**
        $G_R \leftarrow G_R \cup \{\text{Reduce}(g, G \backslash \{g\})\}$
    return G_R

2.3.4 Optimizations

The complexity of Buchberger's algorithm is shown by Mayr [54] to be doubly exponential ($\mathcal{O}(2^{2^n})$) in the number of variables in $\mathcal{R}[\vec{x}]$ and polynomial in the total degree of the polynomials of the input set. Mayr and Meyer [55] demonstrated a worst-case scenario that for a given number of variables, n, there always exists a set of polynomials $F \subset \mathcal{R}[\vec{x}]$ such that, for all $f \in F$, $\deg(f) \leq 5$ and any Gröbner basis for F contains $\mathcal{O}(2^{2^n})$ many elements. This puts a theoretical lower-bound on the (worst-case) complexity of any Gröbner bases computation algorithm.

We seek to improve the average-case complexity of Buchberger's algorithm and reduce the number of critical pairs that are being analysed by removing any redundant critical pairs and testing some conditions to determine if the normal form of a computation is zero. The following two criteria were introduced and proved by Buchberger [11]:

Lemma 2.59 (Buchberger's First Criterion) Let $\rho_1, \rho_2 \in \mathcal{R}[\vec{x}]$ be two polynomials. If $\text{lcm}(\text{LM}_\sigma \rho_1, \text{LM}_\sigma \rho_2) = \text{LM}_\sigma \rho_1 \cdot \text{LM}_\sigma \rho_2$ — i.e., the leading monomials of the two polynomials are relatively prime — then $\text{NR}_{\sigma,G} \mathbf{S}_{\sigma,\rho_1,\rho_2} = 0$.

Lemma 2.60 (Buchberger's Second Criterion) During the analysis of the critical pair $\langle \rho_1, \rho_2 \rangle \in G \times G$ in Algorithm 2.56, if there exists a polynomial $\rho_3 \in G$ such that $\rho_3 \mid \text{lcm}(\text{LM}_\sigma \rho_1, \text{LM}_\sigma \rho_2)$, and the pairs $\langle \rho_1, \rho_3 \rangle$ and $\langle \rho_2, \rho_3 \rangle$ have previously been considered in the computation of G, then $\text{NR}_{\sigma,G} \mathbf{S}_{\sigma,\rho_1,\rho_2} = 0$. The tuple $\langle \rho_1, \rho_2, \rho_3 \rangle$ is called a *Buchberger triple*.

The two criteria above provide a test for determining before-hand if the result of a critical pair computation would lead to zero — however it is not guaranteed for the result to be non-zero if the test fails. The following algorithm combines these

criteria in Buchberger's algorithm. Assume Criterion1 and Criterion2 are predicates that determine if a critical pair of polynomials satisfies Buchberger's first and second criterion, respectively.

Algorithm 2.61 (Buchberger's Improved Algorithm)
Inputs $F = \{f_1, \ldots, f_k\} \subset \mathcal{R}[\vec{x}] \backslash \{0\}$, returns $G \subset \mathcal{R}[\vec{x}]$ such that $\langle G \rangle = \langle F \rangle$ and G is a Gröbner basis.

```
1  let ImprovedGröbner(F) =
2      let G = F and t = |G|
3      let B = {⟨i, j⟩ | 1 ≤ i < j ≤ t}
4      while B ≠ ∅ do
5          let ⟨i, j⟩ ∈ B
6          B ← B − {⟨i, j⟩}
7          if ¬Criterion1(f_i, f_j) then
8              let s = S_{f_i, f_j}
9              let s' = NR_G s
10             if s' ≠ 0 then
11                 t ← t + 1
12                 let f_t = s'
13                 G ← G ∪ {f_t}
14                 B ← B ∪ {⟨i, t⟩ | 1 ≤ i < t ∧ ¬Criterion2(f_i, f_t, B)}
15     return G
```

As mentioned earlier, the performance of Buchberger's algorithm will always have the worst-case complexity of $\mathcal{O}(2^{2^n})$ even with the aforementioned improvements [55]. In §6 we will see how specific subsets of problems can lead (constructively) to algorithms specialized from Buchberger's algorithm that have better overall complexity.

2.3.5 Examples

We will now revisit the sample ideal from Example 2.50 to see how the reduced Gröbner basis is computed.

Example 2.62 Let

$$f_1 = x^2 + y + z - 1, \qquad f_2 = x + y^2 + z - 1, \qquad f_3 = x + y + z - 1$$

be polynomials in $\mathcal{R}[\vec{x}] = \mathbb{R}[x, y, z]$ and $F = \{f_1, f_2, f_3\}$. We will compute the Gröbner basis for the ideal $I = \langle F \rangle$ with term ordering $\sigma = \geq_{\texttt{Lex}}$ using Algorithm 2.56.

First, the algorithm sets up the basis set $G = \{f_1, f_2, f_3\}$ and the working set $B = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$. Next we will traverse the critical pairs one by one:

- $\langle 1,2\rangle$: Computing the S-Polynomial of f_1 and f_2 leads to: $\mathbf{S}_{f_1,f_2} = 1\cdot f_1 - x\cdot f_2 = x^2+y+z-1-(x^2+xy^2+xz-x) = -xy^2-xz+x+y+z-1$. Since the leading term xy^2 is divisible by $\mathrm{LT}_\sigma f_2 = x$, we have $-xy^2-xz+x+y+z-1 \xrightarrow{f_2} y^4+2y^2z-2y^2+y+z^2-z$ which is not divisible by any other polynomials in G. Thus, we add the new polynomial

  $$f_4 = \mathrm{NR}_G\mathbf{S}_{f_1,f_2} = y^4+2y^2z-2y^2+y+z^2-z$$

  to the basis $G \leftarrow G\cup\{f_4\}$ and add $\langle i,4\rangle$ $i = 1\ldots 3$ to the working set B.

- $\langle 1,3\rangle$: Computing the S-Polynomial and its normal form leads to

  $$f_5 = \mathrm{NR}_G\mathbf{S}_{f_1,f_3} = y^3+y^2z-y^2+yz+z^2-z$$

  which is added to the set G and the related pairs are added to the working set B.

- $\langle 2,3\rangle$: The new polynomial

  $$f_6 = \mathrm{NR}_G\mathbf{S}_{f_2,f_3} = y^2-y$$

  is augmented to G and the working set expanded. At this point we analyse the new pairs.

- $\langle 1,4\rangle$: $\mathbf{S}_{f_1,f_4} = -2x^2y^2z+2x^2y^2-x^2y-x^2z^2+x^2z+y^5+y^4z-y^4$, but $\mathbf{S}_{f_1,f_4} \xrightarrow{f_1,f_3} 0$ and no new polynomial is added.

- $\langle 2,4\rangle$: $\mathrm{NR}_G\mathbf{S}_{f_2,f_4} = 0$.

- $\langle 3,4\rangle$: $\mathbf{S}_{f_3,f_4} = -2xy^2z+2xy^2-xy-xz^2+xz+y^5+y^4z-y^4$, and $\mathbf{S}_{f_3,f_4} \xrightarrow{f_2,f_4} -2yz-z^2+z$, thus

  $$f_7 = -2yz-z^2+z$$

  is added to G. Notice that this normal form does not reduce to 0 even though the leading terms of f_3 and f_4 are relatively prime since the algorithm for normal form computes the remainder by dividing by f_2 first before considering f_3.

- $\langle 1, 5\rangle$: Adding new polynomial to G:

$$f_8 = -\frac{1}{2}z^3 + \frac{1}{2}z$$

- $\langle 2, 5\rangle, \langle 3, 5\rangle, \langle 4, 5\rangle, \langle 1, 6\rangle, \langle 2, 6\rangle, \langle 3, 6\rangle, \langle 4, 6\rangle, \langle 5, 6\rangle$ all reduce to normal form of 0.

- $\langle 1, 7\rangle, \ldots, \langle 6, 7\rangle$ and $\langle 1, 8\rangle, \ldots, \langle 7, 8\rangle$ have normal form $= 0$.

Thus, $G = \{f_1, \ldots, f_8\}$ is a Gröbner basis for F.
□

Example 2.63 Let $G = \{f_1, \ldots, f_8\}$ be a Gröbner basis as computed above. We will compute the minimal Gröbner basis for G using Algorithm 2.57. Set $G_M = \emptyset$ and iterate through each polynomial in G:

- f_1: Since $\mathrm{LM}_\sigma f_2 \mid \mathrm{LM}_\sigma f_1$, this polynomial is discarded.

- f_2: $\mathrm{LM}_\sigma f_3 \mid \mathrm{LM}_\sigma f_2$ and thus f_2 is also discarded.

- f_3: No other polynomials divide the leading term of f_3, therefore the monic polynomial f_3 is added to the set G_M.

- f_4: $\mathrm{LM}_\sigma f_5 \mid \mathrm{LM}_\sigma f_4$.

- f_5: $\mathrm{LM}_\sigma f_6 \mid \mathrm{LM}_\sigma f_5$.

- f_6: The monic polynomial f_6 is added to the set G_M.

- f_7: The monic polynomial

$$f_7' = yz + \frac{1}{2}z^2 - \frac{1}{2}z$$

  is added to the set G_M.

- f_8: The monic polynomial

$$f_8' = z^3 - z$$

  is added to the set G_M.

Thus, $G_M = \{f_3, f_6, f_7', f_8'\}$ is the minimal Gröbner basis for G. □

Example 2.64 Let $G_M = \{f_3 = x + y + z - 1, f_6 = y^2 - y, f_7' = yz + \frac{1}{2}z^2 - \frac{1}{2}z, f_8' = z^3 - z\}$ be a minimal Gröbner basis as computed above. We will compute the reduced Gröbner basis for G_M using Algorithm 2.58. Set $G_R = \emptyset$ and iterate through each polynomial in G_M:

- $\text{supp}(f_3) = \{x, y, z\}$ and no leading terms in G_M (which are $\{x, y^2, yz, z^3\}$) divides the elements in support of f_3, therefore no reduction is needed and f_3 is added to the set G_R.
- $\text{supp}(f_6) = \{y^2, y\}$, $\text{supp}(f_7') = \{yz, z^2\}$, and $\text{supp}(f_8') = \{z^3, z\}$ are also not reducible and are directly added to G_R.

Thus, $G_R = \{f_3, f_6, f_7', f_8'\}$ is also the reduced Gröbner basis for G_M. □

David Cox provides some more tutorials of Gröbner bases computations in [23] that are out of the scope of this thesis.

2.4 Applications of Gröbner Bases

Three problems were introduced earlier in this chapter which can be solved by finding (reduced) Gröbner basis of the corresponding set of polynomials, namely:

- §2.2.2: Solving systems of polynomial equations,
- §2.2.3: Determining ideal membership,
- §2.2.4: Finding a confluent polynomial rewrite system.

Given an algebraic problem that is reducible to any of the above questions, we may solve the original problem by finding the Gröbner basis for the reduced problem — which possibly involves interpreting the resultant basis back into the domain of the original problem. In this section we briefly discuss some applications of Gröbner bases as problems that can be reduced to the questions above.

2.4.1 Gaussian Elimination

We can perform Gaussian Elimination on a set of linear polynomials using Buchberger's Algorithm. It can be shown [36, 92] that given a set of linear polynomials

(i.e., the degree of each polynomial is 0 or 1), the resulting basis from Buchberger's Algorithm is also a set of linear polynomials. Furthermore, computing the reduced Gröbner basis on this set leads precisely to the same set of polynomials that Gaussian Elimination provides. Providing a LEX ordering, we can furthermore use elimination theory to back-propagate the variables and solve a system of equations purely using Gröbner bases.

2.4.2 Euclidean Algorithm

Given a set of univariate polynomials (i.e., polynomials of only one variable, $|\vec{x}| = 1$), we can show that the reduced Gröbner basis is a basis of a single element which is the greatest common divisor of all the input univariate polynomials. Moreover, we can show that the result of executing Buchberger's Algorithm on this set performs precisely the steps required to execute the Euclidean algorithm for polynomial gcds [13].

2.4.3 Integer Programming

Another interesting application of Gröbner bases is a special encoding of linear optimization problems as computing the reduced Gröbner basis of a toric ideal as shown by Sturmfels [81]. Hosten [43] shows that the Integer Programming method for linear optimization problems is a special case of Buchberger's algorithm using a special encoding technique.

2.4.4 Graph Colouring

As Loera demonstrated [51], we can implement combinatorial optimization problems by solving systems of polynomial equations. Hillar [41] builds on this theory by choosing a special set of polynomials over the complex field $\mathbb{C}$ that encode a graph colouring problem as a question of finding the Gröbner basis. This also demonstrates that other combinatorial optimization problems can be similarly encoded and solved using Buchberger's algorithm.

2.4.5 Other Applications

There are many more computation problems that can be shown to be special applications of Gröbner bases and Buchberger's algorithm, many of which are highlighted in a series of tutorials by Buchberger and Winkler [13]. Some samples of other such applications are solving boolean satisfiability problems [18, 91], algebraic geometry [76], geometric theorem proving [95] and more.

Chapter 3

Meta Programming

Meta programming is the act of writing programs that manipulate other programs. The primary purpose of meta programming in this thesis is to generate code for the specialized programs and instances of the main algorithm used throughout this thesis: Buchberger's algorithm. Although we will attempt to keep the discussion general within this chapter, we will only focus on the aspects and ideas of meta programming and code generation that are of interest in our implementation of Buchberger's algorithm.

There are three main requirements of code generation within the context of this thesis. We require the generated code to satisfy the following properties:

- Syntactic Correctness: The generated program must be free of syntax errors and be a valid program in the programming language of choice.
- Type Correctness: The generated program must satisfy all the type correctness criteria for compilation. The specialized instances of sub-programs must have compatible data types when composing larger programs, and the type correctness of the composed program must be checked during each operation.
- Semantic Correctness: Generated programs must not only be compilable and valid in the selected programming language, but they must also satisfy a certain contract which defines the behaviour of the programs during execution.

We will address the first two concerns listed above for syntactically correct, type safe code generation in this chapter. The question of semantic correctness in generated programs can only be partially addressed by automatically carrying the specifications

that a program fragment satisfies during code composition to obtain *evidence* of the correctness for the generated code. This point will be the subject of Chapter 4.

3.1 Meta Programming in F#

Throughout the history of computer programming many different definitions have been given for meta programming — e.g., manipulating strings representing code or handling representations of abstract syntax trees, template meta programming, and multi stage code compilation [72, 77, 90] — but they all share the same common goal: *To manipulate or reason about other programs*[1]. With advancements in the concepts of programming languages we see more tools and machinery available for writing better meta programs. In this thesis we are concerned about *language integrated* meta programming, where the components of a meta program are available constructs inside the language itself and there is no need for a meta language to write such programs. In this case "meta program" is perhaps a misnomer and a suggested name of *multi stage* programming [16, 89] is more apt where the focus is on multiple stages of compilation for obtaining code. We will assume the two terminologies are interchangeable in this chapter.

Language integrated meta programming has three main components as were first introduced in Lisp (see [9] for a full description):

(1) **Quote**: To obtain a representation of the code inside the quotation as a value in the program.

(2) **Eval**: To run the piece of code that a value represents.

(3) **Splice**: To compose different fragments of code together.

The concept of the splice operation is possibly the most complicated of this list and deserves more explanation. The real power of meta programming lies within the splice operation for composing code fragments together as we can modify the code and produce new combinations of code fragments using the splice operation. Moreover, we may perform computations on code, such as variable renaming, partial code compilation [84], or even provide optimizations by replacing specialized routines depending on the context of the code [14, 16].

[1]The "other" program may even be the code of the meta program itself, in the case of self-modifying programs.

Having such power over code generation may lead to some undesirable results where the produced code is meaningless, but with the development of more organized meta programming components integrated into languages such as MetaML [90], MetaOCaml [72], Haskell [77] and F# [86] we have access to more expressive meta programming features, such as static guarantees and type safety of splicing to ensure that the produced code fragments can be compiled.

We have chosen the F# programming language [85] as the programming language of choice for the implementation of the Gröbner bases solver generator program in this thesis.

3.1.1 The F# Programming Language

F# is a hybrid functional/object-oriented programming language from Microsoft Research [85]. The syntax of F# is similar to OCaml [44] with some elements from the C# programming language. We will assume that the reader has some familiarity with functional programming and the basic syntax of F# in this thesis, but we will briefly describe the syntax and mechanism of quotation and evaluation in F# that is required for meta programming.

Don Syme introduced the meta programming components in F# programming language and its uses in the generation of dynamic queries and data parallel programs in his paper "Leveraging .NET Meta-programming Components from F#" [86]. This section will provide an overview of the meta programming elements introduced in Syme's paper with the addition of some basic semantics for quotation and evaluation.

Definition 3.1 (F# Language and Values Domain) Let L be the language of F# as described by MSDN [63], Σ be the set of all possible memory states and variable configurations in the .NET environment, and D be the domain of all values representable by all data types in .NET. This domain of values includes —but is not limited to— all the basic value types (integers, floats, strings, etc.), object values (instances of any class type), all type values (members of the `Type` data type), exception values (special type `Exception`), and function types and representations.

A formal framework for representation and operations on the states will be presented in §4.1.1.

Definition 3.2 (Semantic Valuation in F#) Given an F# expression $e \in L$, let $[\![e]\!] : \Sigma \to D$ be the partial valuation function as defined by the operational semantics

of the programming language and the runtime environment. For a state $\sigma \in \Sigma$, the value of $[\![e]\!]\sigma$ is the (typed) value returned by F# interactive (fsi) when e is evaluated in the current state σ of execution. In the cases of compile-time or syntax errors, run-time errors, uncaught exceptions, and non-termination, the value of $[\![e]\!]\sigma$ is simply undefined.

Example 3.3 Let e_1 be the F# expression **fun** $x \to x + 1$ and σ_1 be an initial state of execution for fsi. Then $[\![e_1]\!]\sigma_1$ is a value of type `int` $\to$ `int` which is the compiled function represented by e_1. Let σ_2 be the same state as σ_1 with the additional value assignment $t \leftarrow\!\shortmid [\![e_1]\!]\sigma_1$ and e_2 be the expression t 5, then $[\![e_2]\!]\sigma_2 = 6$ is the result of applying the above function to argument value 5. This may be represented by the following transcript in fsi:

```
> let t = fun x -> x + 1;;
val t : int -> int
> t 5;;
val it : int = 6
```

The variable t is unassigned in state σ_1, thus the value of e_2 when the current state is σ_1 produces a compilation error and remains undefined.
□

Definition 3.4 (Quotation) A typed *code quotation* in F# is constructed by surrounding a syntactically valid and typed code fragment between `<@` and `@>` symbols. For ease of notation, we will use the symbols $\ulcorner \cdot \urcorner$ instead of `<@`$\cdots$`@>` for quoting syntax in writing. Given a state $\sigma \in \Sigma$ and an expression $e \in L$ such that $[\![e]\!]\sigma$ is defined and has the type $'t$, the quotation $\ulcorner e \urcorner \in L$ is another expression where $[\![\ulcorner e \urcorner]\!]\sigma$ is a value of type $\text{Expr}\langle' t\rangle$. The `Expr` data type represents the syntax of an expression and is implemented as an inductive data type. The expression $\ulcorner e \urcorner$ is sometimes called the *lifted* expression of e.

Example 3.5 Following Example 3.3, let e_3 be the expression $\ulcorner$**fun** $x \to x + 1 \urcorner$, then $[\![e_3]\!]\sigma_2$ is a value of type $\text{Expr}\langle$`int` $\to$ `int`$\rangle$ which represents the syntax of the expression e_3, transcribed as:

```
> <@ fun x -> x + 1 @>;;
val it : Expr<(int -> int)> =
  Lambda (x,
       Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
            [x, Value (1)]))
```

□

Definition 3.6 (Evaluation) As an inverse to the lifting operation, to *evaluate* (or "run") a quoted expression, F# uses the extension method `Eval` as provided by the F# PowerPack [57], which compiles a quoted expression into LINQ [10] computation trees and evaluates it. The evaluation of an expression $e \in L$ is represented by the syntax e.`Eval`() which we will denote by $\mathbf{E}\ e$ for ease of notation. Given a state $\sigma \in \Sigma$ and an expression $e \in L$ such that $[\![e]\!]\sigma$ is an expression of type $\text{Expr}\langle' t\rangle$, the evaluation $\mathbf{E}\ e \in L$ is another expression such that $[\![\mathbf{E}\ e]\!]\sigma$ is an expression of type $'t$ if the evaluation of e is defined.

The combined semantics of quotation and evaluation in F# satisfies the law of disquotation: Given an expression $e \in L$ and a state $\sigma \in \Sigma$, $[\![\mathbf{E}^{\ulcorner} e^{\urcorner}]\!]\sigma = [\![e]\!]\sigma$ whenever both $[\![e]\!]\sigma$ and $[\![\mathbf{E}^{\ulcorner} e^{\urcorner}]\!]\sigma$ are defined. To be more precise:

$$\forall_{e \in L, \sigma \in \Sigma}\ (\exists_{e' \in L, \sigma' \in \Sigma}\ [\![e]\!]\sigma = [\![^{\ulcorner} e'^{\urcorner}]\!]\sigma') \Rightarrow ([\![\mathbf{E}\ e]\!]\sigma \doteq [\![e']\!]\sigma')$$

where the quasi-equality $\doteq$ means the values are equal whenever they are both defined, or both undefined.

Example 3.7 Continuing Example 3.5, let σ_3 be the same state as σ_2 with the additional value assignment $q \hookleftarrow [\![e_3]\!]\sigma_2$ and e_4 be the expression $\mathbf{E}\ q$. Then, $[\![e_4]\!]\sigma_3$ is a function of type `int` → `int` as the evaluation of the quotation above. Additionally the expression $e_5 = (\mathbf{E}\ q)\ 5$ has the same value as the expression e_2 from Example 3.3. The following transcript shows these calculations in fsi:

```
> let q = <@ fun x -> x + 1 @>;;
val q : Expr<(int -> int)> = ...
> q.Eval();;
val it : (int -> int) = <fun:ToFSharpFunc@3053-1>
> (q.Eval()) 5;;
val it : int = 6
```

□

3.1.2 Splicing and Quasiquotations

The quote and eval operations defined earlier provide the necessary mechanism for representing code fragments that are syntactically correct and type-safe. We would

like to be able to compose and modify code fragments in a similar fashion with the same static guarantees as $\ulcorner\cdot\urcorner$ and **E** have. Essentially we need a mechanism for embedding the expression represented by a quotation within a larger quotation such that the syntax of the sub-expression is preserved and a post-composition type checking is performed. F# provides the splicing syntax for performing such code compositions:

Definition 3.8 (Splicing) A *spliced* expression is a specially marked sub-expression within a quotation — represented by the unary operator `%` before the marked expression — such that the value of this sub-expression is of type `Expr`. We will use the symbols $\llcorner\cdot\lrcorner$ to mark a spliced sub-expression for ease of notation. Let $e \in L$ be an expression which has a marked sub-expression e_1, such that $e = \cdots\llcorner e_1\lrcorner\cdots$. Then, given a state $\sigma \in \Sigma$, the value $[\![\ulcorner e\urcorner]\!]\sigma = [\![\ulcorner e'\urcorner]\!]\sigma$ such that e' is obtained by substituting the occurrence of e_1 within e with the expression that represents $[\![e_1]\!]\sigma$. A quotation that includes spliced expression may also be called a *quasiquotation* to be consistent with Quine's terminology of quotation theory [93], especially as it is popular within Lisp literature.

Let $e \in L$ be an expression and $e_1, \ldots, e_n$ be pairwise non-overlapping marked sub-expressions of e. Then the value of the quasiquotation $\ulcorner e\urcorner = \ulcorner\cdots\llcorner e_1\lrcorner\cdots\cdots\llcorner e_n\lrcorner\cdots\urcorner$ is obtained by first syntactically replacing every marked expression $e_1, \ldots, e_n$ within e by (typed) temporary place holder variables[2] $x_1, \ldots, x_n$. Then, this new expression e' is evaluated in the context σ by the usual rules defined by $[\![e']\!]\sigma$, and finally the value of the placeholders $x_1, \ldots, x_n$ in $[\![e']\!]\sigma$ are replaced by the values of $[\![\mathbf{E}\ e_1]\!], \ldots, [\![\mathbf{E}\ e_n]\!]$, respectively. If for any i, $1 \leq i \leq n$, the value $[\![\mathbf{E}\ e_i]\!]$ is undefined or does not have the same type as the placeholder x_i, then the value of $[\![\ulcorner e\urcorner]\!]\sigma$ is also undefined.

Let **S** be simultaneous syntactic substitution as defined by Andrews [4], such that given expressions $x, x_1, \ldots, x_n, x'_1, \ldots, x'_n \in L$, $\mathbf{S}^{x_1,\ldots,x_n}_{x'_1,\ldots,x'_n}x$ is obtained by the replacement of x_i in x with x'_i for $i = 1 \ldots n$.

Then:

$$\forall_{\sigma\in\Sigma}\ [\![\ulcorner e\urcorner]\!]\sigma \doteq \mathbf{S}^{x_1,\ldots,x_n}_{[\![\mathbf{E}e_1]\!]\sigma,\ldots,[\![\mathbf{E}e_n]\!]\sigma}[\![\ulcorner\mathbf{S}^{\llcorner e_1\lrcorner,\ldots,\llcorner e_n\lrcorner}_{x_1,\ldots,x_n}e\urcorner]\!]\sigma$$

where $x_1, \ldots, x_n$ are placeholders in the expression.

[2] In fact, the symbol for these holes in F# is an _. We will refrain from using this notation in the expressions for the purpose of visual clarity.

Example 3.9 The expression **fun** $a\ b \to \ulcorner \llcorner a \lrcorner + \llcorner b \lrcorner \urcorner$ represents a function of type $\text{Expr}\langle\texttt{int}\rangle \to \text{Expr}\langle\texttt{int}\rangle \to \text{Expr}\langle\texttt{int}\rangle$ which splices two quoted expressions of integer type to make another expression representing their sum. Let σ be the state which contains the assignment $f \hookleftarrow [\![\textbf{fun}\ a\ b \to \ulcorner \llcorner a \lrcorner + \llcorner b \lrcorner \urcorner]\!]\sigma_{\mathsf{init}}$ where σ_{init} is some initial state, then $[\![f\ \ulcorner 1 \urcorner\ \ulcorner 2 \urcorner]\!]\sigma = [\![\ulcorner 1 + 2 \urcorner]\!]\sigma$. The following transcript shows the above example at runtime:

```
> let f = fun a b -> <@ %a + %b @>;;
val f : Expr<int> -> Expr<int> -> Expr<int>
> f <@ 1 @> <@ 2 @>;;
val it : Expr<int> =
  Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32),
      [Value (1), Value (2)])
> (f <@1@> <@2@>) = <@ 1 + 2 @>;;
val it : bool = true
```

□

3.1.3 A Compiler Extension

One shortcoming of the splice operation in F# as defined in §3.1.2 is that the spliced sub-expressions are evaluated in the context of the parent quotation, and thus the spliced sub-expressions are not connected to any state changes — e.g., introduction of new variables — that may happen within the quoted expression. Consider the following example:

Example 3.10 Let σ be the state as defined in Example 3.9 which carries the assignment $f \hookleftarrow [\![\textbf{fun}\ a\ b \to \ulcorner \llcorner a \lrcorner + \llcorner b \lrcorner \urcorner]\!]\sigma_{\mathsf{init}}$. Additionally, let e be the expression $\ulcorner \textbf{let}\ t = 1\ \textbf{in}\ \llcorner f\ \ulcorner t \urcorner\ \ulcorner t \urcorner \lrcorner \urcorner$. Then the value of this quasiquotation $[\![e]\!]\sigma$ first evaluates the marked sub-expression in the context σ, $[\![f\ \ulcorner t \urcorner\ \ulcorner t \urcorner]\!]\sigma$ which is undefined due to the variable t not existing in this context. F# has a special built-in mechanism to avoid such staging errors. The following transcript demonstrates this error:

```
> let f = fun a b -> <@ %a + %b @>;;
val f : Expr<int> -> Expr<int> -> Expr<int>
> <@ let t = 1 in %(f <@t@> <@t@>) @>;;
error FS0446: The variable 't' is bound in a quotation but is used as part of a
    spliced expression. This is not permitted since it may escape its scope.
```

□

In order to accommodate such statements, we need to modify the definition of quasiquotation from Definition 3.8 to allow for accessing state between multiple stages. This introduces a new challenge of stage safety for variable access: When is it safe and correct for a spliced sub-expression or an inner quotation to access the variables from outside its scope?

Definition 3.11 (Stage Index) Let $e \in L$ be an expression and let ν be a variable that appears in e. The *stage index* of a usage of ν is the number of (nested) quotations textually surrounding the appearance of ν minus the number of nested splices that contain it.

Theorem 3.12 (Scope Safety) Let $e \in L$ and variable ν be as defined in Definition 3.11. Then the variable ν does not escape its scope within e if every usage of ν has a stage index greater than or equal to the stage index of the declaration of ν. An expression e satisfies the stage safety criterion if no variables in e escape their scope.

Proof Due to Walid Taha [89].
□

The extension to the quotation mechanism that we are contributing is performed by adding variable arguments to the "holes" which carry the spliced sub-expressions. These arguments are the variables from the previous stages that are being used in the current splice. The process has two stages:

(1) **Pickling**: When pickling (serializing) a quotation at compile time, we add a list of arguments to each hole for all he variables that are used within the expression from previous stages. The hole object itself is now a function from a list of variables to an expression[3]. These arguments will be supplied at run time.

(2) **Unpickling**: The unpickling (deserializing) of a quotation is performed during run time by the core libraries. At this stage, all the information for running and evaluating the quotation and the (partial evaluation of) spliced sub-expressions is available. At this stage the quotations library supplies the appropriate variable expressions — which have now been computed — as the arguments to the holes which represent the spliced sub expressions. A final dynamic type test is performed after the evaluation of the splice in order to ensure type safety of the final code fragment.

[3]The holes were originally just expression templates in the official F# release.

The following example demonstrates the usage of this extension.

Example 3.13 Let σ and e be the same state and expression as in Example 3.10. The following transcript shows the value of $[\![e]\!]\sigma$ after installing the compiler extension:

```
> <@ let t = 1 in %(f <@t@> <@t@>) @>;;
val it : Expr<int> =
  Let (t, Value (1),
     Call (None, Int32 op_Addition[Int32,Int32,Int32](Int32, Int32), [t, t]))
```

□

Another contribution that we made to the F# libraries is a pretty-printer for quoted expressions which prints a much more reader-friendly output when generating code fragments. The standard printer for the `Expr` type in F# prints the (parsed) expression tree for a given quotation which is much more verbose. For the remainder of thesis, we will always produce transcripts using the contributed expression printer.

3.1.4 Staged Value Types

Code quotations and the splice operation provide the mechanism for partial evaluation of code. In Example 3.13 we saw that the function f was partially evaluated using the template variable t and all the references to the function call were eliminated. Partial evaluation of code is an important advantage in multi stage programming [52] that reduces the amount of abstraction overhead [16] and code duplication [84]. However, if all the information for a computation of a statement is readily available at the *current* stage of compilation, there is no need for delayed or partial evaluation of code and a full evaluation of an expression can be performed at compile time. In order to support both partial and full evaluation of expressions within the same context, we need to introduce a staged type for the values:

Definition 3.14 (Staged Values) The staged value type is an algebraic data type with contains either *ground* values or *code* values. The ground values encapsulate the data that is available "now" at the current stage of compilation, and the code values encapsulate the data that will be available "later" at a future stage of compilation (or execution).

```
type Value<'a> =
    | V of 'a
    | E of Expr<'a>
```

We may also force future evaluation of a ground term by lifting the value to an expression, or force the evaluation of a delayed expression to be performed now. Note that the GetV function may fail if the value is an expression which is not available for evaluation yet.

```
let GetV = function V a -> a | E a -> Quote.Eval a
let GetE = function V a -> <@ a @> | E a -> a
```

The staged type allows us to make decisions in the code on whether an operation can be performed right now or if we need to create a new expression that will compute the result at a future stage. For example, to form pairs (tuples) or to extract elements from a pair, we can define the following operators that contain the definitions for both the "now" and the "later" values:

```
let Pair = function
    | V a,V b -> V (a,b)
    | a,b -> E <@ %(GetE a),%(GetE b) @>

let Fst = function V a -> V(fst a) | E a -> E <@ fst %a @>
let Snd = function V a -> V(snd a) | E a -> E <@ snd %a @>
```

This definition of a staged value can be extended to define multi-stage constants and operators. Each staged operator contains two implementations of the operation: One that works on ground terms and computes the result now, and another that works on expression terms and composes a compound expression which computes the result later. Any multi-valued operator that receives mixed parameters (i.e. both ground and code values) will need to delay the computation of ground values to a future stage and compose a code fragment from the lifted values.

There are a few special "pipe" operators of F# that we use extensively throughout this chapter and later on in the implementation of the code generator. We will only provide the definitions of these operators as defined in the F# standard library for reference:

```
let (|>) x f = f x
let (<|) f x = f x
let (>>) f g x = g (f x)
let (<<) g f x = g (f x)
```

Definition 3.15 (Constants) Staged *constants* are defined similar to value types. The two methods `GenV` and `GenE` generate a constant from either a ground value or an expression, respectively.

```
type Constant<'a> = Value<'a>
module Constant =
    let GenV v = V v
    let GenE ev = E ev
```

We wrap the utility and generation methods within a module to separate the scope of each method. For example, the two functions defined above will be accessed by calling Constant.GenV and Constant.GenE, respectively.

Definition 3.16 (Unary Operators) Staged *unary operators* are functions of type $\text{Value}\langle'a\rangle \to \text{Value}\langle'b\rangle$ which accept a staged value of type $'a$ and produce a staged value of type $'b$. The generator function requires both the "now" implementation of the function of type $'a \to 'b$ and the "later" implementation of type $\text{Expr}\langle'a\rangle \to \text{Expr}\langle'b\rangle$ and produces the staged unary operator of type $\text{Value}\langle'a\rangle \to \text{Value}\langle'b\rangle$. The two applicator methods produce either of the implementations from a staged unary operator upon request.

```
type UnaryOp<'a,'b> = Value<'a> -> Value<'b>
module UnaryOp =
    let Gen f ef = function
        | V v -> f v |> V
        | E e -> ef e |> E
    let AppV u v = v |> V |> u |> GetV
    let AppE u e = e |> E |> u |> GetE
```

Definition 3.17 (Binary Operators) Similar to unary operators, the staged *binary operators* are functions of type $\text{Value}\langle'a\rangle \to \text{Value}\langle'b\rangle \to \text{Value}\langle'c\rangle$ with a generator function and two related applicators.

```
type BinaryOp<'a,'b,'c> = Value<'a> -> Value<'b> -> Value<'c>
module BinaryOp =
    let Gen f ef = fun x y ->
        match x,y with
        | V v1,V v2 -> f v1 v2 |> V
        | E e1,E e2 -> ef e1 e2 |> E
        | _ -> ef (GetE x) (GetE y) |> E
    let AppV b v w = b (V v) (V w) |> GetV
    let AppE b e f = b (E e) (E f) |> GetE
```

The definitions of the algebraic objects earlier in §2 only required constants, unary and binary operators. We also define a `TernaryOp` type similar to the definitions above which will be utilized in some special operators in the future. Since one goal of this thesis is to have a staged implementation of such algebras, we only define the above four types in this chapter.

Additionally, we provide a lifted `Value` version of the common library functions that we use throughout this project. We will separate the lifted operators in modules grouped by the functionality, in a similar approach to that used by Elsheikh, et. al. in Generative Geometry Kernel [15]. We have currently defined the following modules that provide `Value` operations for various tasks. We will not provide the source code for these modules as their implementation is straight forward:

- `Control` module provides operations for program control and flow, such as conditional statements, loops, variable assignments, etc.
- `Function` module provides the operations for abstracting and applying `Value` functions.
- `Bool` module provides the logical and Boolean-valued operators.
- `Tuple` module provides the operations for constructing or deconstructing pairs of values.
- `Ref` module provides the operations for reference cells.
- `Option` module provides the operations for option types.
- `Array`, `List` and `Seq` modules provide the required array, list and sequence operations, respectively.

Example 3.18 (Tuple operators) As a sample for the lifted operations defined above, we provide the implementation of the `Tuple` module as defined earlier:

```
module Tuple =
    let Fst x = UnaryOp.Gen (fun x -> fst x) (fun x -> <@ fst %x @>) x
    let Snd x = UnaryOp.Gen (fun x -> snd x) (fun x -> <@ snd %x @>) x
    let Pair a b = BinaryOp.Gen (fun a b -> a,b) (fun a b -> <@ %a,%b @>) a b
```

□

3.2 Code Generation

We have addressed the main topics with meta programming in F# to generate code. With the ability to generate code fragments that are syntactically valid and strongly typed with guarantees on syntax correctness, type correctness and variable scope correctness, we can now compose code fragments using our combinators to generate derivative programs.

There are two main advantages to code generation: First, we are able to substitute appropriate specializations of different sub-programs within the code and be able to provide some proof of correctness in the generation of the new program. This matter will be the main subject of discussion in §4. In this case, code generation allows us to produce complete instantiated algorithms which contain all the selected optimization without the execution overhead caused by abstraction. The other advantage of code generation lies within the concept of compile-time beta reductions for partial evaluation using the splicing operation of quasiquotations as per discussion of §3.1.2 and §3.1.3.

Ideally, we would like to have a domain-specific language (DSL) that captures all the required code generation routines in a convenient environment. In most modern functional programming languages (Haskell, OCaml, and F# for instance) we can eliminate the need for an external DSL by using the special features in the language such as the development of custom operators for code combinators and special monads for composing code fragments. This section is dedicated to the code generation mechanism used for generating the instances of Buchberger's algorithm in this thesis.

3.2.1 State and Continuation Passing

Continuation-Passing Style (CPS) [83] is a method of programming where the usual sequential flow of statement execution is replaced by passing a *continuation* as an argument to each method that determines how the execution is to be continued. In essence, the continuation represents the future of a computation. Continuations are specially useful here for using a value (by its name) that has not been generated yet. For example, a piece of generated code in an inner expression may require the program to have an extra parameter or a new variable to be introduced prior to the execution of the code. In this section we will briefly introduce the concept of computing with continuations and threading a state through the code generators

using F#'s "computation expressions" feature.

First we will briefly show the F# equivalent of the CPS programming techniques as it was first shown by Swadi, et al. [84] in OCaml and used by Carette and Kiselyov [16] to generate code for Gaussian Elimination.

Definition 3.19 (StateCPS Function) A *continuation* is a representation of the control state of the program [71]. A continuation is represented as a higher-order function from an intermediate result to the final result. Continuations have the type ($'$value $\rightarrow$ $'$answer) $\rightarrow$ $'$answer. A *continuation with state* function requires an additional state value (see §3.2.2) that is passed through by every intermediate computation. In this case, a CPS function accepts the state and a continuation as parameters and invokes the continuation on the result of the function after the computation is performed. A state/continuation function has the type

```
type StateCPS<'state, 'value, 'answer> =
    'state -> ('state -> 'value -> 'answer) -> 'answer
```

In order to use the state and continuation passing methods, we need to convert all the code generators to the generators of type StateCPS. Generating a piece of code (of type Expr<'t>) from a code generator needs an initial continuation k_0 and an initial state s_0. We temporarily define these initial values as k_0 `=` **fun** $s\ v \rightarrow v$ and s_0 `= []`. Given a generator `gen : StateCPS<_,_,_>` we generate the code by running `gen` $s_0\ k_0$.

Example 3.20 (Let Generation) Generating a **let** statement may be done using the simple function `LetGen` $a\ f$ `=` $\ulcorner$ **let** t `=` $\llcorner a \lrcorner$ **in** $\llcorner f\ \ulcorner t \urcorner \lrcorner \urcorner$. The same generator converted to continuation-passing style is:

```
> let LetGenCPS a = fun s k -> <@ let t = %a in %k s <@ t @> @>;;
val LetGenCPS : Expr<'a> -> StateCPS<'b, Expr<'a>, Expr<'c>>
```

To produce the result similar to Example 3.13, we need a generator for the addition function as well. In this case we will make a simple double generator that adds a value to itself:

```
> let DoubleGenCPS a = fun s k -> k s <@ %a + %a @>;;
val DoubleGenCPS : Expr<int> -> StateCPS<'a, Expr<int>, 'b>
```

And finally, we may combine the two generators to make a new generator. This new generator provides the application of `DoubleGenCPS` as the continuation to the `LetGenCPS` generator:

```
> let gen = fun s k -> LetGenCPS <@ 1 + 2 @> s (fun s' k' -> DoubleGenCPS k' s' k)
    ;;
val gen : StateCPS<'a, Expr<int>, Expr<'b>>
```

The expression $\ulcorner 1 + 2 \urcorner$ is chosen as an example here to demonstrate the avoidance of code duplication [84] by using let-generation. Running the above generator provides the result:

```
> gen s0 k0
val it : Expr<int> = <@ let t = 1 + 2 in t + t @>;;
```

□

The "binding" mechanism that was used in Example 3.20 to construct the `gen` function can be extended into a general `Bind` combinator which sequences the result of one generator to another one. This, in addition to a basic `Return` combinator, forms the basis of a *monad* [60]. Using the *computation expressions*[4] notation in F#, this monadic generator is defined as follows:

Definition 3.21 (StateCPS Monad) The monadic construction of a state+CPS generator consists of a *return* method which lifts a basic expression into a trivial generator and a *bind* method which combines two generators by supplying one generator as the continuation of another. The following code defines the `scps` computation expression which defines a basic domain-specific language for working within this monad:

```
type SCPSBuilder() =
    member x.Return a = fun s k -> k s a
    member x.Bind (m,f) = fun s k -> m s (fun s' k' -> f k' s' k)
let scps = SCPSBuilder()
```

Example 3.22 (Let Generation with Monads) The same generator as Example 3.20 coded using the `scps` monad is demonstrated in the following code. The monadic code is placed in a block surrounded by { and }, preceded by the name of the monad (in this case, `scps`):

```
let gen() = scps {
    let! t = LetGenCPS <@ 1 + 2 @>
    let! r = DoubleGenCPS t
```

[4] A computation expression in F# is an equivalent notion to monads with some extensions as demonstrated in [88].

```
    return r
}
```

Which, after running the generator[5], produces the same result as Example 3.20. The notation **let!**, **return**, ... will be explained in Figure 3.1
□

3.2.2 State Operations

In §3.2.1 we used the empty list [ ] as the initial state for running the generators. In this section we will discuss the approach taken in this thesis to provide the state for the code generators which supports a collection of values with heterogeneous data types and provides methods for extending and searching the open records.

In order to retrieve or store the state, we need two special "generators". The `Fetch` method invokes its continuation with the current state as both parameters, and the `Store` method overwrites the current state with its argument before invoking the continuation. These methods are defined as:

```
let Fetch s k = k s s
let Store v _ k = k v ()
```

We maintain the state that is threaded through all the generators by using a list of named objects. The approach taken in this thesis is similar to property lists [59] and open records [16]. The named records in the state are well-typed and unique, and any operation on a typed record is performed through an accessor/modifier object.

Definition 3.23 (States and Records) A `State` is a list of F# strings and objects that is threaded through every generator in the `StateCPS` monad.

```
type State = list<string*obj>
```

Although the state itself is a sequence of untyped objects, we will only allow access to the records through a strongly typed class that provides the same type safety as open records [16]. A `StateRecord` is a special class for accessing and modifying a named record in the `State` object. An instance of `StateRecord` is parametrized by its name and value type, and contains the methods to find, add, remove, or modify this field from the state:

[5] The `gen` value is defined as a trivial function here to circumvent the monomorphism restriction that F# enforces on non-function values.

```
type StateRecord<'a>(name:string) =
    member sr.Name = name

    member sr.Find (s:State) def : 'a =
        match List.tryFind (fun e -> fst e = name) s with
        | None -> match def with
                  | None -> failwith <| "Failed␣to␣locate␣field␣" + name
                  | Some d -> d
        | Some a -> unbox(snd a)

    member sr.Store (v:'a) (s:State) : State =
        match List.tryFind (fun e -> fst e = name) s with
        | Some _ -> failwith <| "Field␣" + name + "␣is␣already␣present."
        | _ -> (name,box v)::s

    member sr.Remove (s:State) : State = List.filter (fun e -> fst e <> name) s

    member sr.Modify v s = sr.Store v <| sr.Remove s
```

3.2.3 Code Combinators

Example 3.20 introduced special code generators made for generating **let** statements. Ideally, during the construction of a higher-level generator, we would like such basic generators to be already available. These operations are called *code combinators*. This section defines some of the more important elements in the combinator library that is used in this thesis.

Generating new variable names

When generating new variables — such as the **let** statements we encountered earlier — F# produces a new unique identifier for variables which is used internally to bind and differentiate the usages of each variable. The textual name of the variable — such as t in Example 3.20 — are only used for printing and reporting purposes and do not identify the variable during the evaluation. This produces an undesirable artifact to the users: multiple uses of the generators such as `LetGenCPS` produce variables that are unique to the internals of the code generation framework, but appear the same to the users[6].

[6]This artifact is only associated with F#. The MetaOCaml implementation of this combinator is unaffected.

Example 3.24 Consider the following code generator:

```
let gen() = scps {
    let! t1 = LetGenCPS <@ 1 @>
    let! t2 = LetGenCPS <@ 2 @>
    return <@ %t1 + %t2 @>
}
```

which generates the following code fragment:

```
> let q = gen () [] k0;;
val q : Expr<int> = <@ let t = 1 in let t = 2 in (t) + (t) @>
```

At first glance, it may appear to the user that the variable t is redefined in the second **let** binding and the final result of this code, when evaluated, should be 4; however, F# has an internal distinction between the two t variables above, and upon evaluation we see that the `t1` and `t2` variables in the generator are properly defined in the generated code:

```
> Quote.Eval q;;
val it : int = 3
```

□

As the **let** generator is one of the most important combinators in code generation, we utilize the states as shown in §3.2.2 to generate new, indexed, variable names such that when two **let** bindings are nested and use the same variable name, then they are produced with different indexed names.

Definition 3.25 (Let Combinator) The state records with the prefix `vars_` track the usage — and the highest index — of each variable introduced by any bindings. The helper method `GetNewVar` modifies the state and returns a unique indexed name for the requested variable.

```
let GetNewVar(n,s) =
    let cnt = StateRecord<_>("vars_" + n)
    let cv = cnt.Find s (Some <| ref 1)
    let n = n + "_" + (!cv).ToString()
    incr cv
    let s = cnt.Modify cv s
    n,s
```

The **let**-generator uses the above method to generate **let** bindings with unique names. The state that is threaded through the code generator is augmented or modified to track the nested bindings.

```
let Let a = fun s k ->
    let n,s' = GetNewVar("t",s)
    Quote.Let n (a s' k0) (k s')
```

Example 3.26 Revisiting Example 3.24 using the new `Let` combinator, we have:

```
let gen() = scps {
    let! t1 = Let (Return <@ 1 @>)
    let! t2 = Let (Return <@ 2 @>)
    return <@ %t1 + %t2 @>
}
```

which generates the code fragment

```
<@ let t_1 = 1 in let t_2 = 2 in (t_1) + (t_2) @>
```

when executed, this evaluates to the same result as Example 3.24, but the (printed) names of the variables are uniquely identified.
□

Sequencing generators

Code generation often requires a sequencing of code fragments from different (imperative) generators. In F#, the ; operator is used for sequencing such statements. We define two distinct sequencing combinators: One for appending a sequence of generators one after another; and another "prepend" combinator for injecting a code fragment before a block of code — such as initiation routines or declarations.

```
let Sequence a b = fun s k -> k s <@ %a s k0; %b s k0 @>
let Prepend a c = fun s k -> <@ %a s k0; %k s c @>
```

Conditional generators and loops

The other important code combinator used in this project are code control combinators which generate conditional **if**-statements and **while**-loops.

```
let If cd th el = fun s k -> k s <@ if %cd then %th s k0 else %el s k0 @>
let While c b = fun s k -> k s <@ while %c s k0 do %b s k0 @>
```

Generating equivalent of **for**-loops can be generalized to the concept of *iterators*, where a special function iterates on a given code generator. A usage for this iterator is provided in §3.2.4.

```
let Iterate it l f = fun s k ->
    let n,s' = GetNewVar("i",s)
    k s' <@ (%it) (%Quote.Lambda n (fun i -> f i s' k0)) %l @>
```

3.2.4 Code Generation DSL

This section defines the domain-specific language (DSL) that we use for the remainder of this thesis for the code generators.

Definition 3.27 (Code Generator) A *code generator* is a special sub-type of the state/continuation monad which only performs on `Expr` code types and threads the state variables of type `State` through the generator. This type is defined as:

```
type CodeGen<'v,'w> = StateCPS<State,Value<'v>,Value<'w>>
```

A code generator of type `CodeGen<'v,'w>` ultimately generates values of type `Value <'w>` which contain either an expression representing the generated program (i.e., the `E` branch in `Value` type), or the computed value of this code fragment (as the `V` branch). This generator is defined as:

```
let Generate m = m [] k0
```

We utilize more extensions from the computation expressions feature of F# to define this DSL. These expressions are an extension to the monad defined in Definition 3.21 and declare more language primitives specific to the domain of code generation. The primitives of the code generation DSL are outlined in Table 3.1.

Definition 3.28 (Code Generator DSL) The `codegen` computation expression is defined using the `StateCPS` monad from §3.2.1 and the combinators from §3.2.3 as described in Table 3.1:

```
type CGBuilder() =
    inherit SCBuilder()
    member x.Delay f = f()
    member x.Bind (m,f) = Bind m f
    member x.Return a = Return a
    member x.Yield a = Return a
    member x.ReturnFrom a = a
    member x.YieldFrom a = a
    member x.Zero() = Return Unit
    member x.For(l,f) = Iterate Seq.Iter l f
```

```
    member x.While(c,e) = While (c()) e
    member x.Combine(a,b) = Sequence a b
    member x.Using(m,f) = fun s k ->
        let n,s' = GetNewVar("t",s)
        Control.Let n m (fun v -> f v s' k)
let codegen = CGBuilder()
```

Note that this is a rather unconventional usage of the `Using` member in F# computation expressions. In general, the **use** notation introduces the binding of an object which requires to be *disposed* after the execution of the code block. In the case of the code generator in this thesis, however, it is used to indicate a new variable that is local *only* within the context of the code block and is "abandoned" (or possibly reused) afterwards.

Example 3.29 (Reduced Gröbner Bases) In this example we will attempt to code some of the data processing needed for reduced Gröbner bases from §2.3.3. This code generator assumes there is a module named `Polynomial` which contains the operations of polynomial algebra from §2.1.4.

```
let gen (G:Expr<seq<Polynomial>>) = codegen {
    use G' = List.Empty()
    for g in G do
        let n = scalar g (div one (LC g))
        use m = Ref.Ref n
        for p in G do
            yield! IfU (Bool.neq p g)
                (codegen {
                    let r = mod (Ref.Deref m) p
                    yield Ref.Assign m r
                })
        yield List.Add G' (Ref.Deref m)
    return G'
}
```

This code generator uses majority of the features of the DSL we introduced for code generation. To generate the `reduce` function which produces reduced Gröbner bases, we wrap the generator in a function expression using the `Fun` combinator defined in the library:

```
let reduce = gen |> Fun |> Generate
```

which produced the following code transcript:

Notation	Method	Description
`codegen {` ... `}`	`Delay`	Wrap the entire computation expression as a generator function.
let! *t* = *cgen* ...	`Bind`	Combine code generators by binding the result of *cgen* as a monadic variable *t* to be used by any following sequenced generators.
do! *cgen* ...	`Bind`	Combine and sequence *cgen* generator in the same style as **let!**, but does not bind the result to any monadic variables.

return *r*
 	
yield *r*
	`Return`
	
	`Yield`
		Returns (yields) a value of type `'v` as a trivial code generator of type `CodeGen<'v,'w>`.

return! *cgen*
 	
yield! *cgen*
	`ReturnFrom`
	
	`YieldFrom`
		Returns (yields) the code generator *cgen* as the result of the current generator.

use! *t* = *cgen* ...	`Using`	Combine code generators similar to **let!** notation, but introduces the new variable *t* as a **let** binding inside the generated code as well. See comment in Definition 3.28.
$cgen_1$; $cgen_2$	`Combine`	Combines and sequences two code generators by generating a sequencing operation (a semicolon) in between the code result of the other two generators.
for `t` **in** *expr* **do** *cgen*	`For`	Generates a **for** loop which introduces a new variable *t* within the body of the loop — iterating through the sequence represented by code fragment *expr* — and runs the generator *cgen* for each iteration.
while *expr* **do** *cgen*	`While`	Generates a **while** loop which runs the generator *cgen* until the condition in code *expr* evaluates to `false`.
	`Zero`	Generates an empty code of type `unit` for empty clauses — such as **if**-statements which do not have an **else** clause.

Figure 3.1: Syntax of the Domain-Specific Language for Code Generation

```
val reduce : Value<(seq<Polynomial> -> List<Polynomial>)> =
  E <@
    fun g ->
        let t_1 = new List<int> () in
        Seq.iter
            (fun i_1 ->
                let t_2 = ref (scalar ((1N) / (lc (i_1)), i_1)) in
                Seq.iter
                    (fun i_2 ->
                        if (i_2) <> (i_1)
                        then (t_2) := (mod (! (t_2), i_2))
                        else (),
                    g);
                t_1.Add (! (t_2)),
            g);
        t_1
    @>
```

□

Example 3.30 (Power Generation) It is traditional (and somewhat overused) in the code generation literature to produce an example of generating the power function. Given a (commutative) monoid $\mathcal{M} = (M, \epsilon, \star)$ and an integer $p \geq 0$, how do we produce a function that for every $m \in M$ computes the exponentiation m^p in an optimal manner? The following code generator produces a specialized and optimal routine for computing just that:

```
let power e mul p =
    let rec gen p m = codegen {
        if p = 0 then return e
        else if p = 1 then return m
        else
            let p2 = p / 2
            use! r2 = gen p2 m
            let r = mul r2 r2
            if p % 2 = 0 then return r
            else
                return mul m r
    }
    gen p |> Fun |> Generate
```

For example, to produce the twelfth power of any integer, we can generate the function:

```
> let pow12 = power Algebra.ZI.one Algebra.ZI.mul 12;;
val pow12 : Value<(int -> int)> =
  E <@
    fun m ->
        let t_1 = m in
        let t_2 = (m * (t_1 * t_1)) in
        let t_3 = (t_2 * t_2) in
        (t_3 * t_3)
    @>
```

□

Chapter 4

Software Specialization

A major objective of this thesis is to provide the knowledge and the machinery for generating specialized instances of software products. This chapter describes software specifications in both formal and informal settings and methods of modularizing and specializing software according to its specifications. We will assume that the reader has some familiarity with computer programming and logic, but no prior knowledge of software engineering is required for this chapter.

We start this chapter with a non-software example of design in civil engineering and construction to draw parallels with design and implementation of software. The first section defines software specifications in the context of this thesis with a focus on abstractions and refinements of specifications. These definitions are closely related to those provided by the Refinement Calculus [6]. Next, we define the ideas behind software product lines and program families with a discussion of the benefits and drawbacks of generating instances of program families. This section also reviews the methodologies and differences between aspect-oriented programming and feature-oriented programming as highlighted by Apel et al. [5]. The third section in this chapter introduces a different approach to abstraction and idealization of specifications which defines some very important terminology for the remainder of this thesis. Finally, we conclude this chapter by revisiting a classic example of modular decomposition of a software product —KWIC— as described by D.L. Parnas [65] and provide some partial implementation of instances of this program family in the F# programming language.

We will start this chapter by a motivating example of design and implementation process which will be referenced throughout this text for parallels to the concepts

introduced:

Example 4.1 Consider the process of a new construction project where we are given the task of planning and constructing a new building. The following example is a sample process that may take place to complete this project.

We may start this project by collecting ideas and brainstorming on what is required. For example, we may start with immediate aspects of the building such as location, area, height, parking, etc. Then we may iteratively add details to this plan by adding requirements (in words) or objectives to the list as more decisions are made about the building.

The next step of the process is to make the blueprints and the plans for this building. This step requires the designers or architects of the project to interpret the requirements from the previous step in a more formal environment such as detailed floor plans, wiring charts, etc. This process is not necessarily a one-way transformation from requirements to blueprints and there is the potential for dialectic design as more refinements are performed on the blueprints. It is important to note that although every refinement on the blueprint can translate to an added detail on the requirements, the converse is not necessarily true. In essence, the blueprints are closer to the reality of the building than the ideas and requirements in the previous step.

Additionally, the blueprints may be broken down into smaller, modular plans for better understanding and customization of the project. For example, the electrical wiring map and the plumbing plans may be presented as two separate documents where each one can be refined independently into a new revision of the plans. This allows for easier understanding and refinement of the sub-projects, especially if each module is delegated to a different specialist in its respective field. Also note that this breakdown of the project does not necessarily start at this stage of the design: If the systems were separated during the requirements planning phase of the project, then each blueprint would be an interpretation of its respective requirements document, and the additional details provided on the documents may provide separate specializations of the blueprints for each sub-project.

The next step in the design is for the engineers to provide a model of the building by actualizing the plans from the previous step into a computer simulation of the physical environment. Similar to the previous step, many lessons may be learned about the constraints of the building during the modelling phase which can be translated back into refinements of the blueprints and plans, but not every refinement

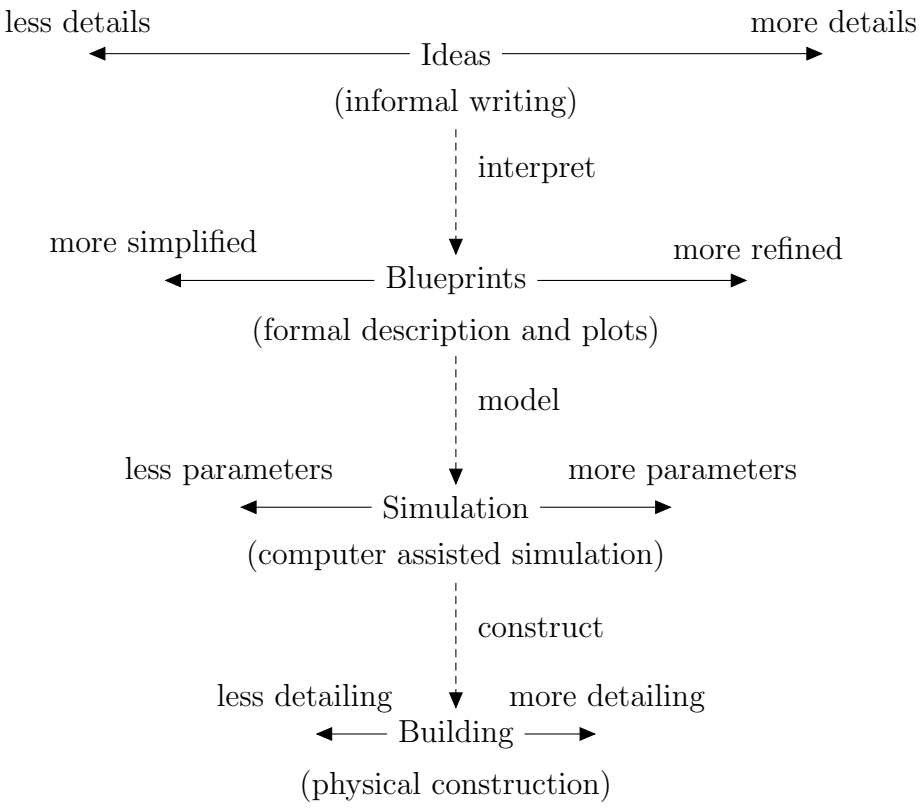

Figure 4.1: Breakdown of a building construction project

of the blueprint can be modelled into the computer simulation as they may not be feasible additions. Again, this model is closer to the reality of the building than the blueprints in the previous step, but still no amount of adding parameters to the model may bring this construction project into physical reality.

Lastly, the actual construction of the building takes place by actualizing the computer simulation and the floor plans from the earlier stages, but similar to the stage transformation from previous steps, this requires the builder to interpret the plans in another domain. Similar to the modular breakdown of the previous stages, this task may also be broken down into separate construction projects of each sub-project. The same concept of specialization of smaller construction projects applies at this level in accordance with the refinements performed on earlier stages of the construction. Figure 4.1 shows the process of this development from pure ideas into a concrete implementation of the building.

□

Remark 4.2 We will highlight the lessons from this example and later in this chapter we will draw parallels between the software design concepts and the ideas in the construction project:

(1) Having the vision of an actual product, we first start the planning by mapping the objectives, requirements and specification of the project in a more conceptual environment. There may be several revisions and improvements to these plans before the implementation.

(2) We gain benefits of easier customization, understanding, and delegation of tasks by breaking down the project into smaller sub-projects where each of these modules can be developed and analysed separately, but they all ultimately join together in the same project.

(3) Within each stage of the project, we have the freedom to add and remove details to and from the task. Removing details helps with making more fundamental decisions and changes to the task and yields a better understanding of the large-scale features. On the other hand, we can iteratively add details to each task to provide a series of refinements on the inner workings of the system.

(4) Each stage is an interpretation of the previous stage in another domain. No amount of refinement or abstraction can transform the task from one domain into another. Each successive stage is an instantiation of the previous stage. Unlike refinement, there is a lower bound to this substantiation where we arrive at the target domain of representation for the project.

(5) Every refinement step may be reinterpreted as a refinement of a model at a higher (more conceptual) stage, but the converse is not necessarily true: Not every refinement can be actualized into a refinement in a more concrete (actualized) stage.

(6) There may be multiple implementations possible for each task, and the choice of which instantiation is better suited at the time of implementation depends on the refinements of the details at a higher, more conceptual stage.

4.1 Software Specification

A software specification is a definition of what a computer program is required to accomplish without defining how it should be done [6]. There are many methods of writing software specifications, such as writing an informal requirements document or design specifications in a semi-formal language such as UML [75]. In this thesis we are only concerned about formal specifications written as logical statements that describe the initial and final states of programs[1].

In this section we will formally define programs and specifications in this context and the meaning of satisfying a specification. Next, we define methods of composing programs and specifications, and finally we define refinements, actualizations, specialization and their relations. Finally, we conclude this section with a detailed example of the function of these concepts and the relationships.

4.1.1 Specifications and Programs

We will formalize the language of writing specifications in a higher-order set-theoretic logic (such as Simple Type Theory [4, 33]) and assume that the reader is familiar with logic and mathematics. Let the logic contain the symbols $\mathbf{T}$ (true), $\mathbf{F}$ (false), $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\Rightarrow$ (implication), $\equiv$ (logical equivalence, which may also be represented at $\Leftrightarrow$), $\forall$ (universal quantification) and $\exists$ (existential quantification).

Definition 4.3 (State) Let V be a (possibly infinite) domain of variable names and D be some domain of values. A *state* σ is a state of computation for variable, expressed as partial assignment of variables from V to values within D. Let Σ be the *state space* containing all such states $V \to D$. States may be defined as assignments of some of the variables written in the set form $\{x := v; \dots\}$ where $x \in V$ is a variable and $v \in D$ is a value. The set of variables in a state σ is represented as $\mathsf{vars}(\sigma) \subseteq V$.

Definition 4.4 (State Valuation) The *state valuation* of a variable x is a function $[\![x]\!] : \Sigma \to D$ such that, given a state σ, $[\![x]\!]\sigma = v$ if σ contains an assignment $x := v$ and is undefined otherwise. Thus, $[\![x]\!]\sigma$ is defined for all $\sigma \in \Sigma$ such that $x \in \mathsf{vars}(\sigma)$.

[1] When concerned with functions, this is analogous to profiling the inputs and outputs of a function.

In fact, states are generalizations (and formalizations) of variable assignments and F# states introduced earlier in Definition 2.25; thus, a variable assignment ψ in some algebra trivially defines a state $\psi \in \Sigma$ as well, and $[\![x_i]\!]\psi = \mathbf{E}^{\psi} x_i$.

A state σ is *contained* in a state σ' if all the variables defined in σ are also defined in σ' and have the same value:

$$\sigma \subseteq \sigma' \equiv \mathsf{vars}(\sigma) \subseteq \mathsf{vars}(\sigma') \wedge \forall_{x \in \mathsf{vars}(\sigma)} \; [\![x]\!]\sigma = [\![x]\!]\sigma'$$

Two states are equal if they assign the same value to each variable; hence $\sigma = \sigma' \Rightarrow \sigma \subseteq \sigma' \wedge \sigma' \subseteq \sigma$.

Definition 4.5 (State Modification) A *state modification* $\sigma[x := v']$ is a new state σ' which contains all the variables $\mathsf{vars}(\sigma) \setminus \{x\}$ as defined in σ while the variable x is assigned the new value v' in the new state σ'. Thus $\mathsf{vars}(\sigma') = \mathsf{vars}(\sigma) \cup \{x\}$, $[\![x]\!]\sigma' = v'$ and $\forall_{y \in \mathsf{vars}(\sigma) \setminus \{x\}} \; [\![y]\!]\sigma = [\![y]\!]\sigma'$. Let $\sigma; \gamma$ be a special state modification where all the variables in γ are renamed by adding a bar on the top and appended to the state σ such that $\sigma; \gamma \equiv \sigma[\bar{x} := [\![x]\!]\gamma \mid x \in \mathsf{vars}(\gamma)]$.

An immediate utility of states is the ability of expressing logical predicates that contain variables within them:

Definition 4.6 (Predicate) A *predicate* p is a boolean expression using the logical constructs and values from D and variables from V such that the value of p is either true or false when all the variables have been replaced by values within D by some state valuation. The set of all predicates is denoted by $\mathbb{P}$.

The valuation of a predicate p is a partial function $[\![p]\!] : \Sigma \to \mathcal{B}$ $(= \{\mathbf{T}, \mathbf{F}\})$ such that for $\sigma \in \Sigma$, $[\![p]\!]\sigma$ is the value of p when any variable x used in p is replaced by its value $[\![x]\!]\sigma$. Note that since states are partial assignments of variables, then $[\![p]\!]$ might also be partial.

Similar to the variables set of a state, $\mathsf{vars}(p)$ for a predicate p is the set of all the variables used in p.

Two predicates p and p' are equivalent, written as $p \equiv p'$, if for all states $\sigma \in \Sigma$, $[\![p]\!]\sigma \equiv [\![p']\!]\sigma$ whenever both $[\![p]\!]\sigma$ and $[\![p']\!]\sigma$ are defined.

Definition 4.7 (Partial Evaluation) Given a predicate p and a state σ, if σ does not contain a definition for every variable used in p — i.e., $\mathsf{vars}(p) \setminus \mathsf{vars}(\sigma) \neq \emptyset$ — then the *partial evaluation* of p in σ is a predicate p' such that $p \equiv p'$ and $\mathsf{vars}(p') \subseteq \mathsf{vars}(p) \setminus \mathsf{vars}(\sigma)$.

Another usage for states is to represent a computer program as a state transformer that can be used for reasoning within a logical framework:

Definition 4.8 (Program) A *program* is a partial state transformer $m : \Sigma \to \Sigma$. Given an *initial state* $\sigma \in \Sigma$ for a machine, $m \cdot \sigma$ is the *final state* of the machine after executing the program m whenever it terminates. The *domain* of m, $\mathsf{dom}(m) \subseteq \Sigma$, is the set of all the initial states σ such that $m \cdot \sigma$ is defined (i.e., m terminates on initial state σ).

Conceptually, a program models the behaviour of a computer program and is typically tied to an implementation, programmed in some computer programming language.

Similar to predicate equivalence, two programs m and m' are *equivalent* if the have the same domain and for all states σ in their domain, $m \cdot \sigma = m' \cdot \sigma$. $\mathsf{vars}(m)$ is defined as the set of variables used in program m.

Definition 4.9 (Function Reification) Let f be a function $f : A \to B$ and $\vec{x}, \vec{y}$ be vectors of variables of type A and B, respectively. Then the *function reification* ${}_{\vec{x}}|f|_{\vec{y}}$ is a program such that for a state $\sigma \in \Sigma$, it evaluates the function f on the value of $\vec{x}$ in the context σ — if it is defined — and returns a new state which assigns the result of the function to the variable $\vec{y}$, i.e.: ${}_{\vec{x}}|f|_{\vec{y}} \cdot \sigma = \sigma[\vec{y} \leftarrow f([\![\vec{x}]\!]\sigma)]$. We may also write this reification as the program $\vec{y} := f(\vec{x})$ when no confusion may arise.

Note that the function reification relies heavily on the fact that the programs are *partial* state transformers, since many mathematical functions cannot be directly translated into computable total programs.

We now have all the required machinery to define specifications and how a program may satisfy a specification. First, we look at a general definition of a specification, and then we define a software specification as a set of the preconditions and postconditions of a state transformer. Specifying a program in terms of its preconditions and postconditions is one of the most popular methods of program specification which is embodied in Hoare logic [42].

Definition 4.10 (Predicate Specification) A *predicate specification* $P \subset \mathbb{P}$ is a set of predicates that define the properties of some software component. A state $\sigma \in \Sigma$ *satisfies* a specification P, written as $\{\!|P|\!\}\sigma$ or $\sigma \models P$, if $\forall_{p_i \in P}\ [\![p_i]\!]\sigma$.

Definition 4.11 (Software Specification) Let $P, Q \subset \mathbb{P}$ be predicate specifications. A *software specification* S — also called a *pre/post-condition specification* — is a statement of the form $P \twoheadrightarrow Q$ where the symbol $\twoheadrightarrow$ is defined as a syntactic separator[2] between the two specifications P and Q. A specification $P \twoheadrightarrow Q$ means that if the precondition specification P is satisfied, then the postcondition specification Q must also be satisfied. The set of all the software specifications is denoted by $\mathbb{S} = \wp(\mathbb{P}) \twoheadrightarrow \wp(\mathbb{P})$.

Given a software specification S, the predicate $\{\!|S|\!\}$ describes whether the specification S is satisfied given an initial state σ and a final state σ', written as $\sigma\{\!|S|\!\}\sigma'$. Let $P \twoheadrightarrow Q$ be a software specification. Then the satisfiability of $P \twoheadrightarrow Q$ with the initial and final states $\sigma, \sigma' \in \Sigma$ is defined as $\sigma\{\!|P \twoheadrightarrow Q|\!\}\sigma' \equiv \{\!|P|\!\}\sigma \wedge \{\!|Q|\!\}(\sigma; \sigma')$ where the ; operation between states is defined as state modification as per Definition 4.5.

We will later see, via Example 4.20, why the postconditions need to be evaluated within the context $\sigma; \sigma'$.

Definition 4.12 Let $S \in \mathbb{S}$ be a software specification, we define the predicates, programs and functions satisfying the specification S by:

- Given an initial state σ and a postcondition predicate q,

$$\sigma\{\!|S|\!\}q \equiv \forall_{\sigma' \in \Sigma}\ \sigma\{\!|S|\!\}\sigma' \Rightarrow [\![q]\!]\sigma'$$

- Given a precondition predicate p and a postcondition predicate q,

$$p\{\!|S|\!\}q \equiv \forall_{\sigma, \sigma' \in \Sigma}\ [\![p]\!]\sigma \wedge \sigma\{\!|S|\!\}\sigma' \Rightarrow [\![q]\!]\sigma'$$

- A program m satisfies $S = P \twoheadrightarrow Q$, $m \models S$, defined as:

$$m \models P \twoheadrightarrow Q \equiv \forall_{\sigma \in \mathsf{dom}(m)}\ \{\!|P|\!\}\sigma \Rightarrow \sigma\{\!|P \twoheadrightarrow Q|\!\}(m \cdot \sigma)$$

- A function $f : A \to B$ satisfies S, $f \models S$, if there exists variables $\vec{x}, \vec{y}$ such that ${}_{\vec{x}}|f|_{\vec{y}} \models S$.

Two software specifications $S, S' \in \mathbb{S}$ are considered equivalent if and only if, for all states $\sigma, \sigma' \in \Sigma$, $\sigma\{\!|S|\!\}\sigma' \Leftrightarrow \sigma\{\!|S'|\!\}\sigma'$. This is often referred to as *observational equivalence*.

Specifications may be *composed* to obtain new —more complex— specifications, such as sequential or parallel definitions of actions.

[2]Note the critical difference between the symbol $\twoheadrightarrow$ and the arrow $\to$ in the text.

Definition 4.13 (Specification Composition) Composition of software specifications is defined inductively as follows:

- A software specification $S \in \mathbb{S}$ is a *composite specification* trivially.
- **(Sequential)** Let S_1, S_2 be composite specifications, $S_1; S_2$ is a *composite specification* that sequences two specifications, defined as:

$$\sigma\{\!|S_1; S_2|\!\}\sigma' \equiv \exists_{\gamma \in \Sigma}\ \sigma\{\!|S_1|\!\}\gamma \wedge \gamma\{\!|S_2|\!\}\sigma'$$

- **(Angelic Choice)** $S_1 \sqcup S_2$ is a *composite specification* that represents a choice for either specification to be satisfied, defined as:

$$\sigma\{\!|S_1 \sqcup S_2|\!\}\sigma' \equiv \sigma\{\!|S_1|\!\}\sigma' \vee \sigma\{\!|S_2|\!\}\sigma'$$

- **(Demonic Choice)** $S_1 \sqcap S_2$ is a *composite specification* that represents an obligation for both specifications to be satisfied, defined as:

$$\sigma\{\!|S_1 \sqcap S_2|\!\}\sigma' \equiv \sigma\{\!|S_1|\!\}\sigma' \wedge \sigma\{\!|S_2|\!\}\sigma'$$

Definition 4.14 (Program Composition) Let m, n be two programs, define $m; n$ to be the program, called the *composition* of m and n, which sequences the execution of n after the execution of m on some initial state, such that $(m; n) \cdot \sigma = n \cdot (m \cdot \sigma)$.

Program composition has some immediate consequences:

Proposition 4.15 Let $f : A \to B$ and $g : B \to C$ be functions, then ${}_x|f|_y; {}_y|g|_z \equiv {}_x|g \circ f|_z$, assuming all three values exist.

Proof For all $\sigma \in \Sigma$,

$$\begin{aligned}
& ({}_x|f|_y; {}_y|g|_z) \cdot \sigma \\
= \ & {}_y|g|_z \cdot ({}_x|f|_y \cdot \sigma) \\
= \ & {}_y|g|_z \cdot \sigma[y \leftarrow f([\![x]\!]\sigma)] \\
= \ & \sigma[z \leftarrow g([\![y]\!](\sigma[y \leftarrow f([\![x]\!]\sigma)]))] \\
= \ & \sigma[z \leftarrow g(f([\![x]\!]\sigma))] \\
= \ & {}_x|g \circ f|_z \cdot \sigma
\end{aligned}$$

$\therefore |f|; |g| \equiv |g \circ f|$.

□

Proposition 4.16 Let $m_1, m_2 : \Sigma \to \Sigma$ be programs and $S_1, S_2 \in \mathbb{S}$ be specifications such that $m_1 \models S_1$ and $m_2 \models S_2$. Then, $m_1; m_2 \models S_1; S_2$.

Proof Assume $m_1 \models S_1$ and $m_2 \models S_2$.

By Definition 4.12 we know that $\forall_{\sigma \in \mathsf{dom}(m_1)}\ \sigma\{\![S_1]\!\}m_1 \cdot \sigma$ and $\forall_{\sigma \in \mathsf{dom}(m_2)}\ \sigma\{\![S_2]\!\}m_2 \cdot \sigma$. Let $\mathsf{dom}(m_1; m_2) = \{\sigma \in \Sigma \mid \sigma \in \mathsf{dom}(m_1) \wedge m_1 \cdot \sigma \in \mathsf{dom}(m_2)\}$ be the domain of $m_1; m_2$. Then $\forall_{\sigma \in \mathsf{dom}(m_1;m_2)}\ (m_1 \cdot \sigma)\{\![S_2]\!\}(m_2 \cdot m_1 \cdot \sigma)$.

Thus, for any $\sigma \in \Sigma$, there exists a $\gamma \in \Sigma$ — namely $\gamma = m_1 \cdot \sigma$ — such that $\sigma\{\![S_1]\!\}\gamma$ and $\gamma\{\![S_2]\!\}m_2 \cdot \gamma$. This satisfies the definition of sequential specifications, then: $\forall_{\sigma \in \mathsf{dom}(m_1;m_2)}\ \sigma\{\![S_1; S_2]\!\}m_2 \cdot m_1 \cdot \sigma$.
$\therefore m_1; m_2 \models S_1; S_2$.
□

We will only use the pre/post-condition method of expressing software specifications in this thesis, and the extension of each definition or statement for composite specifications is assumed to be derived inductively using the constructs in Definition 4.13.

4.1.2 Refinements and Specializations

Let us draw some parallels between the definitions in §4.1.1 and some of the ideas learned from Remark 4.2 on page 72.

The specification for a program is more *conceptual* than the implementation, and a standard approach to software design would be to specify a software product before implementing it. This is a similar approach to 4.2(1). We also learned in 4.2(4) that the stages of the development (such as writing the specifications or the implementation) are connected via an *interpretation* step. The definition of a program satisfying a software specification is similar to this notion, but lacks some details that we will cover in Definition 4.17. We also need to define how to *abstract* a specification to remove some details and how to *refine* a specification to add details similar to 4.2(3). Finally, we explain the relationship between refinements of software specifications that reflect the refinements at a more concrete domain (4.2(5)) and the choice of multiple *specialized* implementations for a given refinement (4.2(6)).

Definition 4.17 (Actualization) Let $S \in \mathbb{S}$ be a specification and $m : \Sigma \to \Sigma$ be a program such that $m \models S$, then m is an *actualization* of S, $m \models\!\!\models S$ — or m

implements S or S *conceptualizes* m — if every state that satisfies the preconditions of S is included in the domain of m:

$$m \models P \rightarrowtriangle Q \equiv \forall_{\sigma \in \Sigma} \{\![P]\!\}\sigma \Rightarrow \sigma \in \mathsf{dom}(m) \wedge \sigma\{\![P \rightarrowtriangle Q]\!\}m \cdot \sigma$$

This notion can further be generalized to cover other interpretations of a specification — such as compiling a program to a lower level language. Although such generalization is interesting, it is out of the scope of this thesis.

Definition 4.18 (Refinement) Let $P, P' \subset \mathbb{P}$ be predicate specifications. P' is a *refinement* of P and P is an *abstraction* of P', written as $P \sqsubseteq P'$, if every state that satisfies P' also satisfies P, i.e. $\forall_{\sigma \in \Sigma} \{\![P']\!\}\sigma \Rightarrow \{\![P]\!\}\sigma$. Let $S, S' \in \mathbb{S}$ be software specifications, then $S \sqsubseteq S' \equiv \forall_{\sigma,\sigma' \in \Sigma} \sigma\{\![S']\!\}\sigma' \Rightarrow \sigma\{\![S]\!\}\sigma'$.

Given a specification $P \subset \mathbb{P}$, there are three fundamental refinements that can be performed. Let $p_1, p_2 \in \mathbb{P}$ be predicates, then adding a new predicate, or adding a conjunction to a predicate, or removing a disjunction from a predicate are all refinements:

(1) $P \sqsubseteq P \cup \{p_1\}$.

(2) $P \cup \{p_1\} \sqsubseteq P \cup \{p_1 \wedge p_2\}$.

(3) $P \cup \{p_1 \vee p_2\} \sqsubseteq P \cup \{p_1\}$.

These special refinements are called *one-step refinements*. Essentially, such refinements embody a single design decision on how a specification is to be implemented. If there exists a sequence of one-step refinements from a specification S to another specification S', then S' is a *stepwise refinement* of S. In general, we will always be working with stepwise refinements in this thesis as the design decisions and implementation of refinements are the focus of software specifications.

Definition 4.19 (Specialization) Let $S = P \rightarrowtriangle Q$ and $S' = P' \rightarrowtriangle Q'$ be software specifications such that $S \sqsubseteq S'$. If any initial and final states that satisfy both S and the precondition P' also satisfy S' entirely, then S' is a *specialization* of S and S is a *generalization* of S', written as $S \trianglerighteq S'$:

$$S \trianglerighteq P' \rightarrowtriangle Q' \equiv S \sqsubseteq P' \rightarrowtriangle Q' \wedge \forall_{\sigma,\sigma'} \{\![P']\!\}\sigma \wedge \sigma\{\![S]\!\}\sigma' \Leftrightarrow \sigma\{\![P' \rightarrowtriangle Q']\!\}\sigma'$$

Let $m, m' : \Sigma \to \Sigma$ be programs and $S, S' \in \mathbb{S}$ be software specifications such that $m \models S$ and $m' \models S'$, then m' *specializes* m on S', $m \trianglerighteq_{S'} m'$, if $S \trianglerighteq S'$. When the choice of S and S' are unambiguous by context, we may simply write this relation as $m \trianglerighteq m'$.

This definition entails that a specialized program may be a replacement for a more generalized one at any point when the precondition assumptions of the refined specification are met. This allows the specializations of a program to exploit the additional information available by the refinement without affecting the satisfiability and the correctness of the original specification. We will extensively use software specializations to generate specialized instances of Buchberger's algorithm in the future chapters.

4.1.3 A Detailed Example

The following example demonstrates the meaning and relations of the concepts introduced in this section.

Example 4.20 Let Ω be the set of all words defined over some alphabet. Given a word $\omega \in \Omega$, denote by $|\omega|$ the length of the word ω, and by ω_i the ith character in ω, where $1 \leq i \leq |\omega|$. The operator $+$ concatenates two words, and the special operation $\omega - \omega_i$ when $1 \leq i \leq |\omega|$ removes the ith character of ω and returns a new word.

An *anagram* of a word is any word which contains all the same characters but not necessarily in the same order. For example, the words "stop" and "spot" are anagrams. A *circular shift* of a word is any word where the same characters appear in the two words in the same order but shifted such that some of the characters from one end appear at the other end. For example, the word "stop" shifted by one character to the left is "tops". The *reverse* of a word is the word which contains all the same characters in the reverse order. For example, the reverse of the word "stop" is "pots".

The following predicates decide if two words define any of the concepts above:

$$\begin{aligned}
\mathit{anagram}(\omega, \omega') &\Leftrightarrow \omega = \omega' \vee \exists_{i,1\leq i\leq|\omega|}\ \omega_i = \omega'_1 \wedge \mathit{anagram}(\omega - \omega_i, \omega' - \omega'_1) \\
\mathit{shift}(\omega, \omega') &\Leftrightarrow \exists_{i,1\leq i\leq|\omega|}(\forall_{j,i\leq j\leq|\omega|}\ \omega_j = \omega'_{j-i+1}\ \wedge\ \forall_{k,1\leq k<i}\ \omega_k = \omega'_{i+k}) \\
\mathit{reverse}(\omega, \omega') &\Leftrightarrow \forall_{i,1\leq i\leq|\omega|}\ \omega_{|\omega|-i+1} = \omega'_i
\end{aligned}$$

Define the following specifications for the programs that compute the words above:

$$\begin{aligned} \mathsf{Anagram} &\equiv \omega \in \Omega \twoheadrightarrow \bar{\omega} \in \Omega \wedge \mathit{anagram}(\omega, \bar{\omega}) \\ \mathsf{Shift} &\equiv \omega \in \Omega \twoheadrightarrow \bar{\omega} \in \Omega \wedge \mathit{shift}(\omega, \bar{\omega}) \\ \mathsf{Reverse} &\equiv \omega \in \Omega \twoheadrightarrow \bar{\omega} \in \Omega \wedge \mathit{reverse}(\omega, \bar{\omega}) \end{aligned}$$

Notice that the symbols ω and $\bar{\omega}$ in the postconditions of each specification refer to the values of variable ω at the initial and the final states, respectively:

$$[\![\omega]\!]\sigma; \sigma' = [\![\omega]\!]\sigma, \qquad [\![\bar{\omega}]\!]\sigma; \sigma' = [\![\omega]\!]\sigma'$$

By the predicate definitions we can deduce that every shifted word and every reversed word is an anagram:

$$\forall_{\omega,\omega' \in \Omega}\ (\mathit{shift}(\omega, \omega') \Rightarrow \mathit{anagram}(\omega, \omega'))\ \wedge\ (\mathit{reverse}(\omega, \omega') \Rightarrow \mathit{anagram}(\omega, \omega'))$$

Similarly, using the specification definitions we can show that the specification for shifts and for reverses refine that of anagrams:

$$\mathsf{Anagram} \sqsubseteq \mathsf{Shift}, \qquad \mathsf{Anagram} \sqsubseteq \mathsf{Reverse}$$

The following F# functions implement the concepts introduced above:

```
let rng = System.Random()
let rec anagram ω =
    if ω = "" then ω
    else anagram(ω.Substring 1).Insert(rng.Next ω.Length, string ω.[0])
let shift ω =
    if ω = "" then ω
    else let i = rng.Next ω.Length in ω.Substring i + ω.Substring(0, i)
let rec reverse ω =
    if ω = "" then ω
    else reverse(ω.Substring 1) + string ω.[0]
```

Then these function provide implementations for the specifications introduced earlier:

$$_\omega|\texttt{anagram}|_\omega \Vdash \mathsf{Anagram} \qquad _\omega|\texttt{shift}|_\omega \Vdash \mathsf{Shift} \qquad _\omega|\texttt{reverse}|_\omega \Vdash \mathsf{Reverse}$$

Moreover, the functions `shift` and `reverse` actualize the specifications for Anagram:

$$_\omega|\texttt{shift}|_\omega \Vdash \mathsf{Anagram} \qquad _\omega|\texttt{reverse}|_\omega \Vdash \mathsf{Anagram}$$

Note that in this thesis we are not concerned with the methods of proving if a given program is an implementation of a specification; however, we will require proving that composing two programs (via code generation) is an implementation of the composite specification. This will be discussed in future sections.

Now let us try some specializations of these software specifications. We know that in general the reverse of a word is not a shift of it, but given a word of length two, it is easy to see that the reverse is also a shift. In fact, when the length of the word $|\omega| = 2$, every anagram of the word ω is a shift. Then we have the following property:

$$\forall_{\omega,\omega' \in \Omega}\ |\omega| = 2 \Rightarrow (\mathit{anagram}(\omega, \omega') \Rightarrow \mathit{shift}(\omega, \omega')) \wedge (\mathit{reverse}(\omega, \omega') \Rightarrow \mathit{shift}(\omega, \omega'))$$

Define the following refinements on the specifications defined earlier:

$$\begin{aligned} \mathsf{Anagram}_2 &\equiv \omega \in \Omega \wedge |\omega| = 2 \ \rightarrowtriangle\ \bar{\omega} \in \Omega \wedge \mathit{anagram}(\omega, \bar{\omega}) \\ \mathsf{Shift}_2 &\equiv \omega \in \Omega \wedge |\omega| = 2 \ \rightarrowtriangle\ \bar{\omega} \in \Omega \wedge \mathit{shifts}(\omega, \bar{\omega}) \\ \mathsf{Reverse}_2 &\equiv \omega \in \Omega \wedge |\omega| = 2 \ \rightarrowtriangle\ \bar{\omega} \in \Omega \wedge \mathit{reverse}(\omega, \bar{\omega}) \end{aligned}$$

Now we can see that $\mathsf{Anagram} \sqsubseteq \mathsf{Shift} \sqsubseteq \mathsf{Shift}_2$, and the property above holds when the preconditions of Shift_2 are satisfied. Thus, we can prove that $\mathsf{Anagram} \trianglerighteq \mathsf{Shift}_2$. Since ${}_\omega|\texttt{anagram}|_\omega \Vdash \mathsf{Anagram}_2$ and ${}_\omega|\texttt{shift}|_\omega \Vdash \mathsf{Shift}_2$, we may also conclude that $\texttt{anagram} \trianglerighteq_{\mathsf{Shift}_2} \texttt{shift}$.
□

4.2 Program Families

One aspect of Remark 4.2 that we have not covered yet is part (2) where we mention easier customization and better understanding of the design process by dividing the project into smaller, more manageable modules[3] or sub-projects. In this section we will briefly introduce the concepts of modular design and the methods of decomposing a software architecture into such modules. A modular design with a flexible architecture and multiple options for each module implementation provides the basis for a program family or a software product line. Later, we will discuss how writing the specifications for each module provides the means of provably correct instances of program families using code generation.

[3]It is important to note the difference between a module as a software unit and the modules as algebraic objects as introduced in §2.

We first start this section by explaining the generic programming techniques for designing software libraries. Then we define modules as software units and describe the differences between a module as a feature against aspect modules. Next, we define the terms for program families and software product lines in the process of software design. Finally, we end this section by describing the methods of modular decomposition that we use in this thesis for designing a family of programs that compute Gröbner bases.

4.2.1 Generic Programming

Generic Programming [26] is the methodology of designing and implementing software by decomposing programs into independent, orthogonal and reusable components. These components are designed and implemented separately and can be arbitrarily combined to produce different variations of software.

We take a generic programming approach to designing and implementing program families. This methodology is similar to that described by A. Stepanov [64] for designing algorithm-oriented generic libraries. In this approach, we analyze the architecture of the algorithm and provide a component-wise (modular) decomposition of the software. These components are designed by identifying various concerns and aspects of the software. Later in §4.2.4 we define the exact criteria for a modular break down of the architecture into orthogonal modules.

Next, we provide the specifications for each module using the methods described earlier in this chapter. The interface of each module must be provided in the most generalized form (as per Definition 4.19) that defines the minimal required behaviour of the component. To provide the most flexible and general form for each method, certain features — such as containers, algebras, etc. — are abstracted away from the components that use them. §4.2.2 describes the differences between these two module types and the method of specifying and parametrizing them.

Finally, we must specify the algorithm such that the specification uses the generalized module interfaces for each module and defines how the (specialized) module implementations are combined in order to produce the implementation of the algorithm.

To design a generic library for an algorithm, we must provide at least one implementation of each module for the use of the generic implementation of the algorithm. Different implementations for the same module must all actualize the specification of

that module. Since the module interface defines the minimal behaviour required to perform the desired operation, each individual implementation is a specialization of the generic module.

4.2.2 Features and Aspects as Modules

There are two popular methods in the literature for creating complex software architectures: Aspect-Oriented Programming (AOP) [45] and Feature-Oriented Programming (FOP) [8]. There is much research on aspects and features of software design which we will refrain from discussing as they are out of the context of this thesis. We will only focus on basic definitions and topics in the two programming paradigms as tools for distinguishing the different types of modules that are presented throughout this chapter and how the ideas of AOP and FOP may work in harmony.

Definition 4.21 (Aspect) An *aspect* is an encapsulation of (sub-)programs which organizes the code related to a specific *concern* in one place. By decomposing a software architecture, we maintain a separation of concerns [19] between the fragments of code which would otherwise be entangled across different sections of the program.

Here, our approach to aspects is much more similar to Dijkstra's original ideas of "separation of concerns" [27, 28] than the modern object-oriented approaches of aspects in code [79, 80].

Definition 4.22 (Feature) A *feature* is a reflection of some architectural requirement or design decision which is embodied as a programming unit. Features contain an implementation of basic ideas and concepts that form the structural workings of the software and are incrementally refined to reflect changes in the design of the software [5].

Features are closer to the specification layer of the software design process than to the implementation layer; in general, modifications to a feature have effects on the overall design and architecture of the software [67] whereas the changes to an aspect are locally contained. This is a generalization of the concept of container access components by Stepanpov [64].

Definition 4.23 (Module) A *module* is a basic building-block of software that organizes the program into separate responsibility assignments [65]. We use modules

to separate the tasks performed in a program to allow the implementation of each module to be (re-)written with little knowledge of the implementation of other modules (regardless of their dependencies). To achieve this, modules are only aware of the specifications and the *interface* of other modules and do not rely on the implementation and inner code of relevant modules.

There are two types of modules:

(1) **Aspect Modules**: These modules encapsulate an aspect as per Definition 4.21 which organizes the relevant code to a specific concern in the design. An aspect module is often a collection of definitions (variables and programs[4]) that share the same underlying concept but are not necessarily required to be in a collection other than for organization and aesthetics purposes. An example of an aspect module is a logging module which contains a collection of tracing, printing and recording methods.

(2) **Feature Modules**: A feature module is a collection of related definitions that form the implementation of a specific feature in the design of the software as per Definition 4.22. Unlike aspect modules, the variables and programs in a feature module are necessary to be presented together and have deep dependencies between them. A modification to an element in a feature module often requires other related concerns in the same feature to be modified as well, and a refinement (addition) to a feature causes global changes to the whole program. An example of a feature module is the implementation of a *ring* structure (Definition 2.4) where any constant and operator in the structure is only useful when the entire ring module is present.

In general, a module may share a common set of information that is local to the functions in that module only and is "hidden" from the external modules. Accessing or modifying this "secret" is only possible through the methods provided by the module, and any changes to this secret only has local effects on the implementation of the module and any external access to this information is unchanged. The concept of information hiding was first proposed by Parnas [68] and we will use this concept extensively in the design of the code generator in this thesis.

[4]In object-oriented programming, a module variable is typically called a *field* and a module program is called a *method*.

A software architecture is always divided into modules of either type. In §4.2.4 we will discuss the methods and the criteria for this decomposition. We split the concept of a "module" into two different aspects: an interface and an implementation.

Definition 4.24 (Module Interface) The *interface* of a module is a collection of specifications that govern the behaviour of the module members (variables and programs) and their relationship to the module's secret. The variables in the module are specified as a pair of a name and a predicate specification that uses this variable. The programs in the module are similarly defined as pairs of names and software specifications. Let $\mathbb{M}$ be the set of all module interfaces. A module interface is written as follows:

moduleinterface:
variable_1: $\mathit{spec}_1 \in \mathbb{P}$
$\vdots$
variable_n: $\mathit{spec}_n \in \mathbb{P}$
program_1: $\mathit{softspec}_1 \in \mathbb{S}$
$\vdots$
program_m: $\mathit{softspec}_m \in \mathbb{S}$

The variable specifications in a module interface are called the *invariants* of the module. A program specification $P \rightarrowtriangle Q$ in a module interface has the *covariant* specification P and the *contravariant* specification Q.

Definition 4.25 (Module Implementation) The *implementation* of a module is a collection of variables and programs that satisfy the specifications provided in the interface of the module. Given an interface $I \in \mathbb{M}$, let $\text{Inv}(I) \in \mathbb{P}$ be the conjunction of all the invariants $\mathit{spec}_1 \wedge \cdots \wedge \mathit{spec}_n$. Let M be a module which contains the variables $M.\text{variable}_1, \ldots, M.\text{variable}_n$ and the programs $M.\text{program}_1, \ldots, M.\text{program}_m$. Then M *implements* (or actualizes) the interface I, written as $M \models I$, if:

$$\forall_{\text{program}_i \in I}\ M.\text{program}_i \models I.\mathit{softspec}_i \wedge \forall_{\sigma \in \Sigma} \{\![\text{Inv}(I)]\!\}\sigma \Rightarrow \{\![\text{Inv}(I)]\!\} M.\text{program}_i \cdot \sigma$$

Designing the software architecture by defining the interface for modules has the advantage that implementations of modules are replaceable as long as they actualize the same interface. Since the other modules in the software only rely on the specification of the methods and not on their specific implementation, a replacement for a specific module does not require a reassembly of the software product. Thus, if we

can prove that a specialization of a program (within a module) is possible due to the refined specifications, then the module can be replaced by its specialization without affecting the correctness of any of the other modules and programs. We will use this technique to specialize the modules in this thesis.

Definition 4.26 (Parametrized Module Interface) In special cases, a module interface may depend on other (already-defined) module interfaces to specify some variables or programs. A module interface $I \in \mathbb{M}$ is *parametrized* by the module interfaces $I_1, \ldots, I_m \in \mathbb{M}$ if the statement of any of the variable or program specifications in I requires some implementation of these modules. We use the following notation to specify a parametrized module interface:

moduleinterface $M_1, \ \ldots, \ M_k$:
$M_1 \Vdash I_1 \in \mathbb{M}$
$\vdots$
$M_1 \Vdash I_k \in \mathbb{M}$
variable_1: $spec_1 \in \mathbb{P}$
$\vdots$
variable_n: $spec_n \in \mathbb{P}$
program_1: $softspec_1 \in \mathbb{S}$
$\vdots$
program_m: $softspec_m \in \mathbb{S}$

The specifications $spec_1, \ldots,$ $softspec_1, \ldots$ may use any variables $M_1.\text{variable}_1, \ldots$ or programs $M_1.\text{program}_1, \ldots$ in their statement.

4.2.3 Program Families

The terms "program family" and "software product line" commonly appear in the software engineering literature when modularization or generation of programs are topical. In this section we address the differences between the two terminologies in the context of this thesis and define the program family that will later be utilized in §5.

It is possible — either by code generation or by manual linking — to produce software by assembling a program together from a common pool of modules. A library or toolkit is a collection of modules — both in interface and implementations of them — which provides the functionality for producing programs. Essentially, when a library is "complete" within a certain context, it provides the framework for

producing any programs which fall within that very specific context; the only coding required from the programmer is choosing the specific implementations within that framework and perhaps some of the necessary "piping" between the modules. This forms a very domain-specific framework for production of software, which is commonly known as a *software product line*. Carnegie Mellon University's Software Engineering Institute (SEI) [78] defines:

> A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

The concept of a software product line can then be further specialized into a specific line of products which share both the same set of common modules and features, and also the same goal in the outcome of the products. In this case, all the generated products from the SPL are essentially different implementations of the same program which only differ in the specifics of their implementation and in possible refinements in the common specification. This tightly bound product line generates a family of products:

Definition 4.27 (Program Family) A *product family* [66] is a group of software products that share the same set of common properties defining the behaviour of each instance within the family. Each member of the family is a specific product from an SPL which satisfies all the common properties of the family; thus, it is beneficial to describe and study this common specification first before analysing each individual instance of the family.

It should come as no surprise to the reader that the ultimate goal of this research is to produce a family of products which all compute Gröbner bases, but differ vastly in their choice of modules and implementation depending on the properties of the input data and the details required of the output data. Ideally, a choice of a "simpler" set of input polynomials or requesting less information in the output should simplify the (generated) member of the family both in terms of time and space complexity. For example, knowing that the set of input polynomials is always linear (i.e., of degree 1) should simplify the generated solver to perform Gaussian Elimination more efficiently compared to the average-case performance of Buchberger's algorithm.

4.2.4 Modular Decomposition of Software Architectures

When creating the architecture for a software product, it is often helpful to start with a more generic, less detailed plan of the product and refine or actualize the design to the specifications for a specific instance of the product, similar to the plan described in Example 4.1. To properly design the architecture of a software product, we first define a generic version of the product that encompasses all possible members in the product family [26]. This generic "super product" may then be specialized into each instance within the family.

At this point the lesson learned from 4.2(2) may be applied to properly breakdown the architecture of the software into more manageable modules that separate the concerns in the products into disjoint programming units. The specifications for the interface of these modules, which can either be an aspect module or a feature module, defines a *modular decomposition* of the software architecture. D.L. Parnas [65] first described the criteria for determining the module structures and their relationships. According to Parnas, the modular design of software has the following benefits:

- Managerial: Development time should be shortened because separate groups would work on each module with little need for communication
- Product Flexibility: it should be possible to make drastic changes to one module without a need to change others
- Comprehensibility: it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

As we learned from 4.2(6), each interface for these modules may have multiple implementations which actualize it. Assuming that there exists a proof (either by formal means or as an informal proof) that the specifications for an implementation is a specialization of the specifications in the interface[5], then we have the ability to replace the implementation of each module in the architecture as long as the specialization is applicable.

We aim to be producing specialized instances of Buchberger's algorithm by a proper decomposition of its architecture into modules, and then applying the fine-tuned specializations of each module that are only applicable in the context of a

[5]A module specification is a specialization of another module if every invariant and program specification is a specialization of the respective specification in the interface.

specific instance to generate a new member of the program family for Gröbner bases solvers. This (prescription of) proof for the specialization of the modules provides a partial proof of correctness for the generated member of the family by removing the task from the code generator and instead relying on the machinery defined in §4.1 to infer the correctness of the finished product.

Much in the same way as product families in manufacturing factories (such as automobiles, etc.), it is possible for a specific member of the family to not contain an implementation for a module that is not used by it — or more precisely, replacing the implementation of a module by a "dummy" version.

Example 4.28 (KWIC: KeyWord-In-Context) The KeyWord In Context example introduced by Parnas [65] is an often-used example for software design and architecture. Although the description and implementation of the problem is quite simple, the main advantage of using this example is the wide variety of architectural and design possibilities for implementing the problem [38, 74, 94].

A KWIC program reads an input of text (in this example we specifically talk about the titles of the books, journals, or articles within some index) and makes all the circular word shifts for each title. Then, it sorts the shifted titles for quick indexing and searching. Finally, it returns the sorted index of all the shifts as some form of output.

A *circular word shift* — similar to the definition of `Shift` in Example 4.20 — is defined as all the possible shifts of the words (as opposed to the letters) in a phrase. Given a title (alternatively, a sentence or a phrase), we can view this phrase as a list of words that construct it. For example, the title of the book (and TV series) "A Game of Thrones" is broken down into an ordered list of four words. By performing all the circular shifts on the words in this title, we obtain four possible representations for this book: "A Game of Thrones", "Game of Thrones A", "of Thrones A Game", and "Thrones A Game of'. When searching for the title of the book, any of the four titles mentioned above should refer to the same book (with the possible addition of commas in the title for clarity).

The KWIC system is broken down into four main processing modules:

(1) `Input`: Reads, parses, and stores the input data.

(2) `Shift`: Makes circular shifts of each title.

(3) `Sort`: Sorts all the shifted titles alphabetically for indexing.

(4) `Output`: Produces the output of the indexes for consumers of the system.

In addition, there is often an implicit `Control` module that connects these modules and the data flow; however, it is possible to have an explicit controller as we will see in the sample implementations. The control module is responsible for sequencing of the calls involving the other modules and handling errors in case they are not directly addressed by the other modules.

These processing modules correspond exactly to the major processing steps taken during the execution of the program as one may even draw them on a flowchart. Essentially, this decomposes the architecture of KWIC into five distinct aspect modules. A model using only these modules suffers from a major architectural challenge: What happens when design decisions change? Such as the choice of data structure used, the input and output formats, or even online sorting of the index on demand. For some of these changes — such as a data structure change — all of the processing modules now need to be re-implemented to accommodate the changed architecture. We can abstract the choice of data structures and storage as feature modules to avoid this.

For some other changes — such as changing the behaviour of `Sort` to perform on-demand partial sorting — we need to alter the behaviour of the modules to be more flexible and general in the way that data flows in the software. In this case we only need to generalize the specification of the modules to allow such specializations between different implementations.

Some implementations of KWIC require storage for their data — this is an evolution to the `Line Storage` module defined in the enhanced decomposition of the KWIC problem by Parnas. For example, the `Input` module might be saving the titles in a database for the `Shift` module to operate on. Thus, some additional modules are required that might need to be present in certain implementations:

(1) `Store`: The storage for the strings in KWIC index.

(2) `Shifts`: The storage for the circularly shifted strings.

(3) `Sorted`: The storage for the sorted strings.

Notice that the method of storage is decided by each specific design — whether the strings in `Shifts` and `Sorted` are stored as full text or references to previous indexes is open to implementation, and the modules `Shift` and `Sort` need not know about these

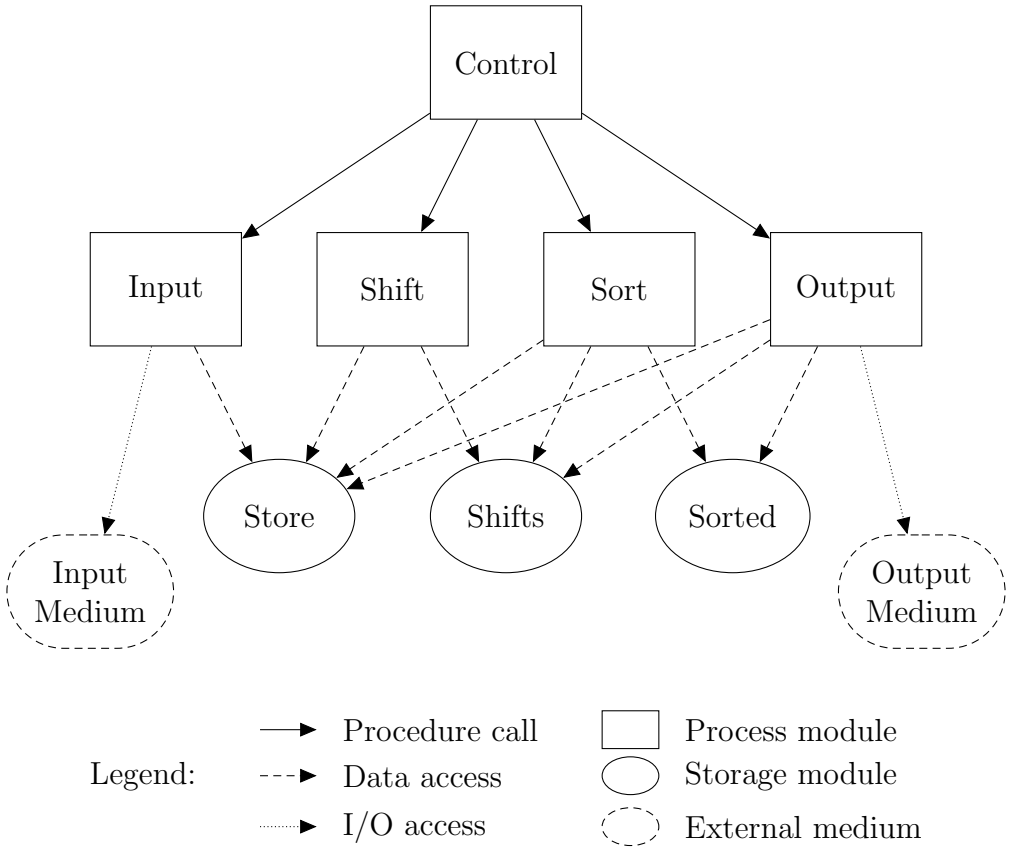

Figure 4.2: General architecture of the KWIC indexing program

decisions. In this case the choice is *hidden* from higher level processing modules. A basic architecture of the KWIC problem is presented in Figure 4.2.

There are many design decisions to be made when implementing the KWIC example, such as: Is the data stored, or passed between modules? How would the data be stored, as text, or indexes? How do modules communicate with each other? Should all of the data be processed at start, or only on demand? What are the types of data being processed? And other such questions. There are four main architectures proposed for this problem [38, 74]:

(1) Shared data model

(2) Data abstraction model

(3) Event based model

(4) Pipes and filters model

All of these models can be obtained using different choices for each module or connection from Figure 4.2. For example, by making some of the connections trivial and storage as simple data pipes, one could arrive at a design that looks like Figure 4.3.

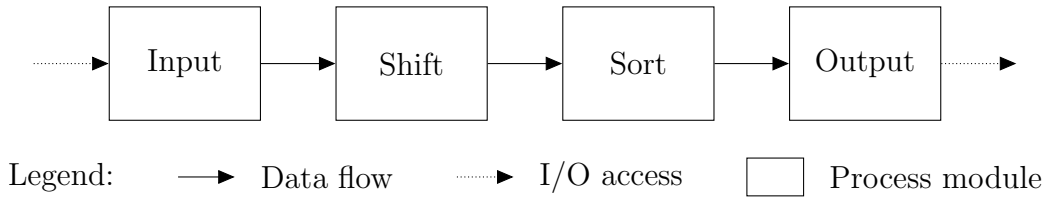

Figure 4.3: The "pipe and filter" specialization of the KWIC architecture

In order to avoid making a new implementation of the system for every design change, we created an experimental software product line for generating implementations (members of family) of KWIC programs according to the design decisions and options that we choose.

There are many more possible specializations of this architecture. For example, KWIC has been implemented as a meta program by Michael VanHilst and David Notkin as C++ templates [94].

□

Chapter 5

The Generative Approach to Algebraic Computations

Having carefully defined the mathematical background needed to analyse Gröbner basis solving algorithms and computational algebra in §2, we now turn to the software architecture design methodology described in §4 to decompose Buchberger's algorithm into a modular design similar to the KWIC example (Example 4.28) and design a generic interface and specifications for any specializations of Buchbeger's algorithm[1]. We can then implement these modules as F# types — an `interface` for each module interface and an `object` type per module implementation — to construct a framework for implementing solvers for computational algebra problems. Finally, we apply the code generation DSL as defined in §3 to construct a generator which produces any instance of the program family for solving Gröbner basis problems. This code generator is parametrized by all the modules in the decomposition of Buchbeger's algorithm and the base modules in the algebra that define the working environment. We also provide some implementations of each module to generate some sample programs.

In §5.1 we carefully define the module interfaces for all the algebraic objects and polynomial types as defined in §2.1. These provide the main feature modules in the code generator for this thesis, and any other modules performing algebraic computations are parametrized by the algebra modules. Any changes to the choice of these modules (such as those for a base ring structure or ordering) makes fundamental changes to the inner workings of the generated program.

[1]Some of the more important specializations were highlighted in §2.4.

Next, in §5.2 we provide a decomposition of Buchbeger's algorithm into aspect modules (which are, in turn, parametrized by the algebra feature modules) with careful generalization of the specifications to allow generation of many specializations of Buchberger's algorithm.

Finally, §5.3 defines the code generator (in the `codegen` DSL) which implements this product family. This generator is, in fact, very similar to the pseudo-code for Buchberger's algorithm in §2.3, but can be highly specialized through code generation to produce any instance of the special sub-algorithms. Showing that any implementation of the aforementioned modules is a specialization of the generic interface, we can provide a partial proof for the correctness of the generated instance.

5.1 Algebraic Objects

Recall the definitions of basic algebraic objects from §2.1. In this section we define the interfaces and implementations of the feature modules that embody these objects. We start the design process with a *carrier* object, which consists only of a set, and refine this interface to define monoids, groups, etc. The MathScheme library [53] contains an extensive definition of algebraic objects — most of which are out of the scope of this project. We take a similar approach in this section and define the algebraic objects both axiomatically as specifications and programmatically as type definitions.

Defining the algebraic feature modules in this manner requires an incremental refinement of modules, starting from a carrier set and refining in small steps to the more complex structures such as fields or algebras. We will take the approach of little theories as defined in §5.1.1 to define these.

Next, to be able to implement the algebraic modules in F# we require the programming techniques to define both the interfaces and the implementations of these modules in such a way that the implementations are open-ended types and contain at least the functionality dictated by the interface, as well as taking advantage of the typing system in F# to define the refinement relation between modules as object inheritance. §5.1.2 defines the concepts of parametric types and *shapes* using object expressions in F#.

Next, we apply these techniques to program the interfaces for the algebraic feature modules in §5.1.3 and provide some module implementations of the most common usage for these types.

Finally, §5.1.5 follows the same footsteps to define the interface of the objects for polynomial algebras — i.e., monomials, terms and polynomials — as parametrized modules over the base algebra types. We also provide a module implementation of a generic multivariate polynomial algebra using both dense and sparse terms.

5.1.1 The Little Theories Method

The little theories method [34] uses a network of theories to formalize a portion of mathematics by "small" step refinement relations between theories. In the little theories method, every theory (or module interface, in case of the definitions used here) is an extension (or refinement, respectively) of some prior theory defined in the library — with the obvious exception of the base object types. In contrast, an ad-hoc "big theory" method requires a full definition of every theory (interface) as a part of a single big theory (interface) with no inheritance relation.

In the little theories method, we define every interface by a series of one-step refinements. These refinements consist of either an addition of a new member (variable or program), or a refinement to the specification of a previously-defined member. These refinements are all related to natural algebraic extensions of the base types that are being refined or joined.

For example, Figure 5.1 shows the incremental little theory refinements from a definition of a carrier set to a monoid and further on to a group, where each refinement step adds the respective values and functions that extend the theory of one algebraic object to the next. In this figure we only define the types of the operations that are added, and we assume that the reader can infer what axioms are added from the definitions of these objects given earlier.

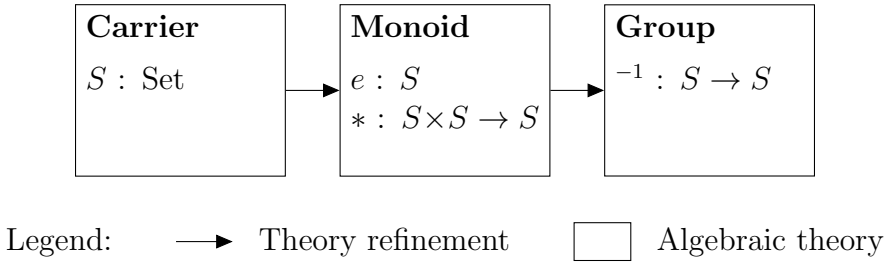

Figure 5.1: Basic algebra objects as little theories

More complex interfaces are then constructed by either refining or joining dif-

ferent module interfaces as required. We seek to design the interfaces for all the algebraic objects necessary in this chapter using the little theories method for easier categorization and organization of the modules.

5.1.2 Parametric Types and Shapes

As mentioned earlier, the feature modules introduced in this chapter form a basis for all the other modules that utilize the computational algebra library. In this case, every aspect module (and in fact, some of the more complex feature modules) are *parametrized* by the types defined by one or more feature modules. For example, the aspect module which is in charge of maintaining the working set and critical pairs of polynomials in Buchberger's algorithm is dependent on the type of polynomials and the operators defined in the polynomial algebra module. Furthermore, the selection of critical pairs may also be dependent not only on the polynomial algebra, but on the choice of term ordering as well.

The generic programming features [62] of F# provide a simple mechanism for parametrizing interfaces and module implementations by generic types. Additionally, it is possible for the implementation of a module to contain more definitions than the provided interface — as long as the required variables and programs are present and satisfy the type and specification requirements. In this case, every implementation of the module is an instance of an unnamed class which satisfies the module's interface. We utilize the *object expressions* [40] syntax in F# to define this specific construction of module implementations. Such object interfaces are called *shapes*. See Hardin and Rioboo [1] for earlier uses of this technique.

To demonstrate the usage of shapes and parametric types, we define a simple interface and implementation for ordered sets.

Interface 5.1 (Order) Given a set S, an ordering relation is a reflexive, antisymmetric, transitive binary relation $\geq\colon S \times S \to \mathbb{B}$ on the elements of the set S. We define the module interface for an ordered set parametrized over the choice of the underlying set by introducing a `cmp` binary operator which performs both comparison (an order relation) and equality (an equivalence relation) on the set S. Given an ordering relation $\geq$, the following interface represents the comparison operation:

Order S:

$_{x,y}|\mathrm{ge}|_b\colon\ x \in S \wedge y \in S \ \twoheadrightarrow\ \bar{b} \in \mathbb{B} \ \wedge\ \bar{b} \Leftrightarrow x \geq y$

For practical programming purposes, we use an altered version of this interface with a standard `cmp` function where `cmp(x,y)` is a negative number when $x < y$ (i.e. $\neg x \geq y$), zero when $x = y$ ($x \geq y \wedge y \geq x$) and positive when $x > y$:

Order S:
$_{x,y}|\text{cmp}|_c$: $x \in S \wedge y \in S \;\; \twoheadrightarrow$
$\bar{c} \in \{-1, 0, 1\} \;\wedge$
$(\bar{c} \geq 0 \Leftrightarrow x \geq y) \;\wedge$
$(\bar{c} \leq 0 \Leftrightarrow y \geq x) \;\wedge$
$(\bar{c} = 0 \Leftrightarrow x \geq y \wedge y \geq x) \;\wedge$
$\text{cmp}(y, x) = -\bar{c} \;\wedge$
$\forall_{z \in S} \;\; \text{cmp}(y, z) = \bar{c} \Rightarrow \text{cmp}(x, z) = \bar{c}$

The last two lines of the above specification contain redundant clauses that can be derived from the earlier clauses. Later in §5.1.4 we will show how such statements can aid the implementation of a module.

Assuming that the type `'s` defines a carrier set, the `Order` interface is defined as the F# type:

```
type Order<'s> =
    abstract cmp: BinaryOp<'s,'s,int>
```

We implement this parametric interface using an object expression which uses the `comparison` type in F#, which provides a `compare` method for values of type `'t` when `'t: comparison`:

```
let GenOrderFromType<'t when 't: comparison> =
    { new Order<'t> with
        member o.cmp = Compare
    }
```

This defines a new unnamed class implementing the `Order` interface, and constructs a new object of this type, which is in turn returned as the result of the order generator function above. `Compare` is a `BinaryOp` (binary operator as per Definition 3.17) defined in our libraries which calls the standard comparison function when a type `'t` implements the interface `comparison`.

Note that although the antisymmetry and transitivity of the ordering relation is specified in the interface, we were not able to reflect this specification in the implementation. We will discuss this in more detail in §5.1.4.

5.1.3 Interfaces for Algebraic Objects

We now possess all the required skills, techniques and the machinery for defining the algebraic objects defined earlier as F# module interfaces. Figure 5.2 shows the module breakdown of the base algebraic objects. These feature modules are defined similarly to the design used in Mei [98] for forming a module system for mechanized mathematics.

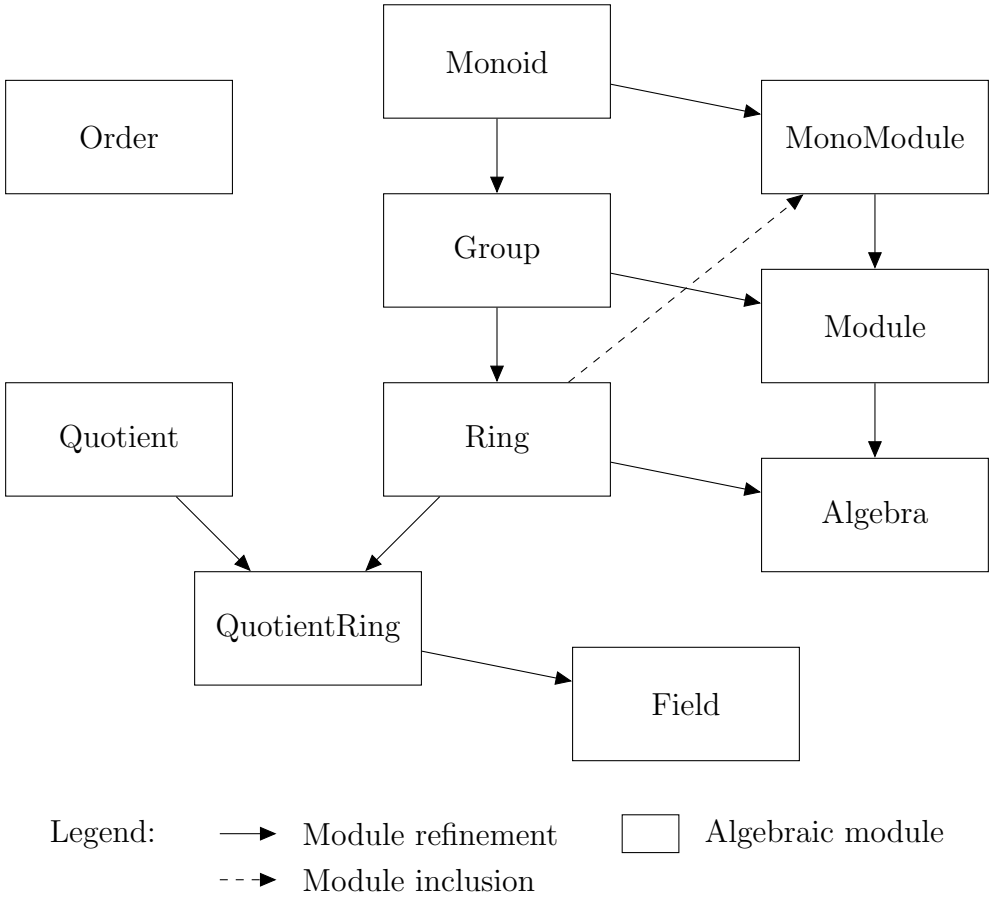

Figure 5.2: Interfaces and relations for algebraic objects

Interface 5.2 (Monoid) We start the algebraic objects by defining the interface of the `Monoid` module as an additive monoid. The monoids are defined additively (with 0 and $+$) as opposed to multiplicatively (with 1 and $*$) for practical reasons in the code, as the additive monoid is used more often as a parent interface for rings than its multiplicative counterpart.

Monoid S:
  zero: $\text{zero} \in S$
  ${}_{x,y}|\text{add}|_z$: $x \in S \wedge y \in S \;\rightarrowtriangle$
    $\bar{z} \in S \;\wedge$
    $\text{add}(y, x) = \bar{z} \;\wedge$
    $\forall_{w \in S}\; \text{add}(\bar{c}, w) = \text{add}(x, \text{add}(y, w)) \;\wedge$
    $x = \text{zero} \Rightarrow \bar{z} = y \;\wedge$
    $y = \text{zero} \Rightarrow \bar{z} = x$

Similar to the case of antisymmetry and transitivity of the `Order` module, the commutativity and associativity of the method `add` in the monoids will not be implemented. This is for practical reasons since F# does not possess the necessary machinery for verification of these properties. This property may be feasible to define in the future with the help of theorem proving additions to the .NET framework such as Z3 [25].
The interface for the `Monoid` feature module is defined as the F# type:

```
type Monoid<'s> =
    abstract zero: Constant<'s>
    abstract add: BinaryOp<'s,'s,'s>
```

Note that the axioms for associativity and commutativity of the `add` operator are not implemented here since the F# type system is not powerful enough to express these requirements.

Interface 5.3 (Group) The first refinement relation between the algebra modules is the introduction of inverse elements to monoids which produces a group structure. We define the specification for the `Group` object as a refinement of the `Monoid` object as follows:

Group $S \;\sqsupseteq$ Monoid S:
  ${}_{x}|\text{neg}|_y$: $x \in S \;\rightarrowtriangle$
    $\bar{y} \in S \;\wedge$
    $\text{add}(x, \bar{y}) = \text{zero} \;\wedge$
    $x = \text{zero} \Rightarrow \bar{y} = \text{zero}$
  ${}_{x,y}|\text{sub}|_z$: $x \in S \wedge y \in S \;\rightarrowtriangle$
    $\bar{z} \in S \;\wedge$
    $\text{add}(x, \text{neg}(y)) = \bar{z} \;\wedge$
    $x = y \Rightarrow \bar{z} = \text{zero} \;\wedge$
    $x = \text{zero} \Rightarrow \bar{z} = \text{neg}(y) \;\wedge$
    $y = \text{zero} \Rightarrow \bar{z} = x$

The subtraction operator is not a necessary definition in group theory as it can be defined in terms of addition and negation operators. We chose to include a separate definition of the `sub` operator in the structure here for both presentation and optimization purposes, as a distinctly generated subtraction operator can be specialized and optimized independently from other operations.

The syntax in F# for expanding a module interface is:

```
type Group<'s> =
    inherit Monoid<'s>
    abstract neg: UnaryOp<'s,'s>
    abstract sub: BinaryOp<'s,'s,'s>
```

The inheritance relation between the `Monoid` and `Group` interfaces allows a type-safe static casting between the generated module implementations. Hence, any code generator or parametrized modules that require the presence of a `Monoid` object could automatically use any `Group` structures as well since we can prove that `Monoid` $\sqsubseteq$ `Group`.

Note that in all of the F# interfaces above (`Order`, `Monoid` and `Group`) only the type of the variables and programs are defined that reflect the type inclusion clauses of the specifications. The rest of the specifications are left "open" as the F# type system is not powerful enough to define these as a part of the definition. We address the enforcing of these clauses in the generators for these module implementations in §5.1.4.

None of the methods specified so far were at their minimal specification form. One may rightfully claim that many of the clauses in the specifications — such as the last three lines in the `sub` specification for groups — are not necessary and can be derived from the rest of the specification. We are including these "special" cases in the interface for these modules and the implementations may take advantage of these results during code generation. For example, when the implementation for the `sub` method of some group has the information available at compile time (i.e. the `V` branch of the `Value` type) that y is, in fact, zero, then it is able to completely avoid the generation of any subtraction operation within the code and simply return the code fragment $\ulcorner x \urcorner$.

Interface 5.4 (Ring) The `Ring` module is defined as a refinement of the `Group` modules using the same axioms defined in Definition 2.4. Recall that after defining ring objects in Definition 2.4 we established that throughout this thesis we will only use finitely generated commutative rings with identity.

Ring S $\sqsupseteq$ Group S:
  one: $\text{one} \in S \ \wedge \ \text{one} \neq \text{zero}$
  $_{x,y}|\text{mul}|_z$: $x \in S \wedge y \in S \ \twoheadrightarrow$
    $\bar{z} \in S \ \wedge$
    $\text{mul}(y, x) = \bar{z} \ \wedge$
    $\forall_{w \in S} \ \text{mul}(\bar{c}, w) = \text{mul}(x, \text{mul}(y, w)) \ \wedge$
    $x = \text{one} \Rightarrow \bar{z} = y \ \wedge$
    $y = \text{one} \Rightarrow \bar{z} = x$
    $x = \text{zero} \vee y = \text{zero} \Rightarrow \bar{z} = \text{zero}$
  $_{x,p}|\text{pow}|_y$: $x \in S \wedge p \in \mathbb{N} \wedge \neg(x = \text{zero} \wedge p = 0) \ \twoheadrightarrow$
    $\bar{y} \in S \ \wedge$
    $x = \text{zero} \Rightarrow \bar{y} = \text{zero} \ \wedge$
    $x = \text{one} \Rightarrow \bar{y} = \text{one} \ \wedge$
    $p = 0 \Rightarrow \bar{y} = \text{one} \ \wedge$
    $p > 0 \Rightarrow \bar{y} = \text{mul}(x, \text{pow}(x, p - 1))$

```
type Ring<'s> =
    inherit Group<'s>
    abstract one: Constant<'s>
    abstract mul: BinaryOp<'s,'s,'s>
    abstract pow: BinaryOp<'s,int,'s>
```

Defining the quotient ring object from Definition 2.34 requires the introduction of a new object for quotients. These objects do not necessarily define a new algebraic type, but are used as a "bundle" of refinement relations and method definitions which add *long division* (i.e. division with remainder) to other structures. In the sense of aspect modules, this is a separation of all the methods that are concerned with division, remainders, greatest common divisors, least common multiples and crossfactors.

Interface 5.5 (Quotient Ring) The `QuotientRing` module implements a quotient ring object from Definition 2.34 that adds the concepts of division with remainder, gcd, lcm, and τ from Definition 2.13 to any ring structure:

QuotientRing S $\sqsupseteq$ Ring S:
  $_{x,y}|\text{longdiv}|_{d,r}$: $x \in S \wedge y \in S \ \twoheadrightarrow$
    $\bar{d} \in S \wedge \bar{r} \in S \ \wedge$
    $x = \text{add}(\text{mul}(y, \bar{d}), \bar{r}) \ \wedge$
    $\text{longdiv}(\bar{r}, y) = (\text{zero}, \bar{r}) \ \wedge$
    $x = \text{zero} \Rightarrow \bar{d} = \text{zero} \wedge \bar{r} = \text{zero} \ \wedge$
    $y = \text{zero} \Rightarrow \bar{d} = \text{zero} \wedge \bar{r} = x \ \wedge$
    $y = \text{one} \Rightarrow \bar{d} = x \wedge \bar{r} = \text{zero} \ \wedge$

$$
\begin{aligned}
&\quad x = y \Rightarrow \bar{d} = \text{one} \wedge \bar{r} = \text{zero} \\
&{}_{x,y}|\text{div}|_d\colon\ x \in S \wedge y \in S \ \rightarrowtriangle \\
&\quad \bar{d} \in S \cup \{\text{None}\} \ \wedge \\
&\quad \bar{d} \neq \text{None} \Leftrightarrow \text{longdiv}(x, y) = (\bar{d}, \text{zero}) \\
&{}_{x,y}|\text{rem}|_r\colon\ x \in S \wedge y \in S \ \rightarrowtriangle \\
&\quad \bar{r} \in S \ \wedge \\
&\quad \exists_{d \in S}\ \text{longdiv}(x, y) = (d, \bar{r}) \\
&{}_{x,y}|\text{gcd}|_g\colon\ x \in S \wedge y \in S \ \rightarrowtriangle \\
&\quad \bar{g} \in S \ \wedge \\
&\quad \text{div}(x, \bar{g}) \neq \text{None} \wedge \text{div}(y, \bar{g}) \neq \text{None} \ \wedge \\
&\quad \forall_{g' \in S}\ \text{div}(x, g') \neq \text{None} \wedge \text{div}(y, g') \neq \text{None} \Rightarrow \text{div}(g, g') \neq \text{None} \\
&{}_{x,y}|\text{lcm}|_l\colon\ x \in S \wedge y \in S \ \rightarrowtriangle \\
&\quad \bar{l} \in S \ \wedge \\
&\quad \text{div}(\bar{l}, x) \neq \text{None} \wedge \text{div}(\bar{l}, y) \neq \text{None} \ \wedge \\
&\quad \forall_{l' \in S}\ \text{div}(l', x) \neq \text{None} \wedge \text{div}(l', y) \neq \text{None} \Rightarrow \text{div}(l', l) \neq \text{None} \\
&{}_{x,y}|\tau|_\tau\colon\ x \in S \wedge y \in S \ \rightarrowtriangle \\
&\quad \bar{\tau} \in S \ \wedge \\
&\quad \text{mul}(\bar{\tau}, x) = \text{lcm}(x, y) \ \wedge \\
&\quad \text{mul}(\bar{\tau}, \text{gcd}(x, y)) = y \ \wedge
\end{aligned}
$$

As we mentioned earlier, it is more convenient programmatically to abstract out the quotient operations into a separate aspect module and refine other interfaces such as quotient rings, fields, monomial monoids and polynomial algebras by means of inheritance.

```
type Quotient<'s> =
    abstract longdiv: BinaryOp<'s,'s,'s*'s>
    abstract div: BinaryOp<'s,'s,'s option>
    abstract rem: BinaryOp<'s,'s,'s>
    abstract gcd: BinaryOp<'s,'s,'s>
    abstract lcm: BinaryOp<'s,'s,'s>
    abstract τ: BinaryOp<'s,'s,'s>
```

Finally, appending the `Quotient` aspect module to the `Ring` feature module leads to the new `QuotientRing` module which contains the specification above as the rings where the Euclidean algorithm (division with remainder) is possible:

```
type QuotientRing<'s> =
    inherit Ring<'s>
    inherit Quotient<'s>
```

The next logical step is to expand the definition of a quotient ring to have guaranteed fraction-free division, which also leads to unique inverses.

Interface 5.6 (Field) The `Field` module refines the `QuotientRing` module by adding unique multiplicative inverses to the non-zero elements of the ring. This leads to a refinement of the specification for the `div` program to always have zero remainders. Additionally, the `pow` exponentiation program is refined in a way to define negative integers in the exponent:

Field $S \sqsupseteq$ QuotientRing S:

$_x|\text{inv}|_y$: $x \in S \backslash \{\text{zero}\} \rightarrowtriangle$
$\bar{y} \in S \ \wedge$
$\text{mul}(x, \bar{y}) = \text{one} \ \wedge$
$x = \text{one} \Rightarrow \bar{y} = \text{one}$

$_{x,y}|\text{div}|_z$: $x \in S \wedge y \in S \backslash \{\text{zero}\} \rightarrowtriangle$
$\bar{z} \in S \ \wedge$
$\text{mul}(x, \text{inv}(y)) = \bar{z} \ \wedge$
$\text{longdiv}(x, y) = \bar{z}, \text{zero}$

$_{x,p}|\text{pow}|_y$: $x \in S \wedge p \in \mathbb{Z} \wedge \neg(x = \text{zero} \wedge p = 0) \rightarrowtriangle$
$\bar{y} \in S \ \wedge$
$x = \text{zero} \Rightarrow \bar{y} = \text{zero} \ \wedge$
$x = \text{one} \Rightarrow \bar{y} = \text{one} \ \wedge$
$p = 0 \Rightarrow \bar{y} = \text{one} \ \wedge$
$p > 0 \Rightarrow \bar{y} = \text{mul}(x, \text{pow}(x, p-1)) \ \wedge$
$p < 0 \Rightarrow \bar{y} = \text{inv}(\text{pow}(x, -p))$

In the F# definition of the `Field` interface, the `pow` method does not require any signature refinements; however, the `div` method's signature is redefined to reflect the changes to the remainder-free division algorithm. In this case the `Field` module contains two separate interfaces for the `div` method: one that is inherited from refining the quotient ring interface, and a new one that reflects the new field division.

```
type Field<'s> =
    inherit QuotientRing<'s>
    abstract div: BinaryOp<'s,'s,'s>
    abstract inv: UnaryOp<'s,'s>
```

The definition of an R-MonoModule (Definition 2.8) and similar composite structures require a type inclusion of an external ring of coefficients R into another algebraic structure. We achieve this by parametrizing the module in the type of both of the involved objects. The following interfaces demonstrate this technique:

Interface 5.7 (R-MonoModule) An R-`MonoModule` M is parametrized by the ring of coefficients R and the monoid M containing the base elements:

MonoModule $R, M \sqsupseteq$ Monoid M:
coefficients: coefficients $\models$ Ring R
${}_{s,x}|\text{scalar}|_{y}$: $s \in R \wedge x \in M \twoheadrightarrow$
$\bar{y} \in M \wedge$
$s = \text{coefficients.one} \Rightarrow x = \bar{y} \wedge$
$\forall_{r \in R}\ \text{scalar}(\text{coefficients.mul}(r, s), x) = \text{scalar}(r, \bar{y}) \wedge$
$\forall_{r \in R}\ \text{scalar}(\text{coefficients.add}(r, s), x) = \text{add}(\text{scalar}(r, x), \bar{y}) \wedge$
$\forall_{z \in M}\ \text{scalar}(s, \text{add}(z, x)) = \text{add}(\text{scalar}(s, z), \bar{y})$

```
type MonoModule<'r,'m> =
    inherit Monoid<'m>
    abstract coefficients: Ring<'r>
    abstract scalar: BinaryOp<'r,'m,'m>
```

Interface 5.8 (R-Module) An R-`Module`[2] refines both the `MonoModule` and `Group` modules to impose an additional group structure on the base elements, but does not refine either the ring of coefficients or the scalar multiplication operator.

Module $R, G \sqsupseteq$ MonoModule $R, G \cup$ Group G

```
type Module<'r,'g> =
    inherit MonoModule<'r,'g>
    inherit Group<'g>
```

In Definition 2.10, we defined an R-Algebra S using a structural homomorphism from the ring of scalars to the ring of elements that lifts a member of R into domain of S. In order to write a specification for this feature module, we first need to define a homomorphism axiomatically. Unfortunately, at this time we do not have the required machinery in F# to axiomatically define a homomorphism as per Definition 2.6 and thus we will leave the specification of homomorphism open at this point and define this object as a pure code module.

Interface 5.9 (Homomorphism) A `Homomorphism` is defined as a mapping from an algebra (module type) O_1 to algebra (module type) O_2 with the carrier sets (types) S_1 and S_2, respectively. Additionally, a `Homomorphism` object also contains objects from the structures O_1 and O_2 for reference.

[2]The font distinction between the words "`Module`" and "Module" plays a significant role in some of the definitions.

```
type Homomorphism<'s1,'s2,'o1,'o2> = {
    s: 'o1
    d: 'o2
    φ: UnaryOp<'s1,'s2>
}

type Endomorphism<'s,'o> = Homomorphism<'s,'s,'o,'o>
```

Interface 5.10 (R-Algebra) Similar to an R-`Module`, an R-`Algebra` S refines both the `Module` and `Ring` modules and imposes an additional ring structure on the base elements. Moreover, an embedding homomorphism (the structural homomorphism) between the rings R and S is added to the specification which lifts the elements from the scalars domain to the elements domain. The scalar multiplication is refined to reflect this embedding:

$$
\begin{aligned}
&\text{Algebra } R, S \sqsupseteq \text{Module } R, S \cup \text{Ring } S\text{:}\\
&\quad \text{embedding: embedding} \models \text{Homomorphism (Ring } R \to \text{Ring } S)\\
&\quad {}_{s,x}|\text{scalar}|_x\text{: } s \in R \wedge x \in S \twoheadrightarrow\\
&\qquad \bar{x} \in S \wedge\\
&\qquad \text{mul}(x, \text{embedding}.\varphi\ s) = \bar{x} \wedge\\
&\qquad x = \text{coefficients.one} \Rightarrow x = \bar{x} \wedge\\
&\qquad \forall_{r \in R}\ \text{scalar}(\text{coefficients.mul}(r, s), x) = \text{scalar}(r, \bar{x}) \wedge\\
&\qquad \forall_{r \in R}\ \text{scalar}(\text{coefficients.add}(r, s), x) = \text{add}(\text{scalar}(r, x), \bar{x}) \wedge\\
&\qquad \forall_{y \in M}\ \text{scalar}(s, \text{add}(y, x)) = \text{add}(\text{scalar}(s, y), \bar{x})
\end{aligned}
$$

```
type Algebra<'r,'s> =
    inherit Module<'r,'s>
    inherit Ring<'s>
    abstract embedding: Homomorphism<'r,'s,Ring<'r>,Ring<'s>>
```

5.1.4 Implementing Algebra Modules

In this section we show how each algebraic interface defined above is implemented in such a way that the specifications of the respective module are *enforced* on the implementation of each method.

In all the module specifications defined in §5.1.3 we only specified the properties of the algebraic objects (both in variables and in programs) which can be programmatically enforced in F# either by some static guarantees (such as type checking) or dynamic guards (such as sanity checks and conditional evaluation). We may break

down the clauses in the variable/program specifications into the following seven criteria and implement each one accordingly:

(1) **Type Refinements**: These module refinements appear in the declaration of the module and specify the base modules that are refined and extended to make this new object type. The type refinements commonly appear in the form

$$module \sqsupseteq module_1 \cup module_2 \ldots$$

and are implemented as interface inheritance in the code. The inheritance relation between these types also guarantees safe static up casting within the instances of the module.

(2) **Type Guarantees**: These clauses in the specification bind the type of a specified variable to a specific object type. Although in the formal specification these clauses are posed as set membership assertions such as

$$variable \in type$$

they are carefully specified such that they are directly translated to type constraints in F#. The signature of the function or the type of the value in the code implements this type of specifications.

(3) **Type Inclusion**: Type inclusions were first demonstrated in Interface 5.7 for the ring of coefficients. This type of specification has the form of

$$variable \Vdash module$$

and generally binds one or more of the type parameters to have an algebraic structure defined in another module. The specified variable in this case contains an instance object which implements a certain module interface. For example, in the case of the `MonoModule` interface, the `coefficients` variable contains an implementation of the `Ring` object which defines the ring structure on the type (set) R. The type inclusion specifications are implemented by binding the type of the variable as the interface for the included module — possibly parametrized by one of the module's own type parameters.

(4) **Value Restrictions**: Restricted values for variables (especially function parameters) are a special form of type guarantees which not only bind the type

of a variable to a specific type, but also restrict the domain to disallow certain special cases. This refinement to the type guarantee specification generally has the form

$$variable \in type \backslash \{value_1, value_2, \dots\}$$

and is implemented similarly to the type guarantees specification by defining the signature of the function (or the type of the value) and an additional run-time dynamic check on the value of this parameter (or variable). A simple example of such restrictions is the exclusion of zero as the denominator in quotients. Implementation 5.12 demonstrates the implementation of value restrictions in the `div` method of fields.

(5) **Grounded Value Conditions**: The conditional specifications impose some additional value properties or define new behavioural properties when certain predicates are satisfied. The value condition clauses in specifications generally have the form

$$variable = value \;\Rightarrow\; property_1 \;\wedge\; property_2 \dots$$

where the equality condition could simply be replaced by other comparison predicates such as $\geq$, etc. These specifications are enforced in the implementation at execution time by conditional guards such as **if** statements or pattern matching. Implementation 5.11 demonstrates the (additive) identity specifications imposed on monoid structures in the implementation of the `add` method.

(6) **Universal Equality**: These specifications define the behaviour of a program (method) in terms of other binding properties of some other programs and values and have the general form of

$$\forall_{variable \in type}\; property_1 \;\wedge\; property_2 \dots$$

where the quantified properties depend on the bound variable. In general, these specifications cannot be implemented in F#. We attempt to *not* define such statements in the specifications of the modules when the implementation is not possible (or infeasible). Requirements such as commutativity or associativity of the operators are examples of universal statements that will not be enforced. Implementation 5.14 demonstrates the implementation of universal

equality for the `scalar` function in an R-Algebra as defined per specifications of Interface 5.10.

(7) **Value Profiles**: Similar to the universal binding specifications defined above, the value profiles enforce a certain relation between two properties of the specification. Value profiles have the following form:

$$property_1 = property_2$$

where the equality could be replaced by other comparison predicates such as $\neq$ or $\geq$ in a similar fashion to value condition specifications. These specifications are often restrictive statements about properties of some values and are generally not enforceable within the F# code. The relationship between the negations `neg` and the `add` function within `Group` is an example of a value profile that cannot be implemented in code, but it presented in the specifications due to its important nature in the definition of the operations. Other profiles such as the relationship between `div` and `inv` functions within a field is partially implementable in the code when only one of the method implementations is present. Implementation 5.12 shows how this is implemented for the field division when an implementation of `inv` method is provided.

Before providing any implementations, we need to warn the reader about the **module** keyword in F#. A **module** in F# is merely a collection of types, variables and functions that are only bound together by a common namespace. Although this is not very different from our definition of an *aspect* module in this thesis, we will *not* be using F#'s **module**s to provide any module implementations. Instead, we are using these entities to organize the generators and miscellaneous methods that are related to our own modules. In the following implementation, the method `Monoid.Gen` generates an instance of type `Monoid<'m>`. At this point, the font difference between the words "**module**", "`Module`" and "module" may play an important semantic role in the meaning of any text or code.

Implementation 5.11 (Monoid Generator) This example shows the generation of a monoid object that implements the Monoid module specified in Interface 5.2 and the corresponding F# interface for it. Given two functions `zero` and `add` such that `zero` $\models$ Monoid.zero and `add` $\models$ Monoid.add, the `Gen` function produces an object of type `Monoid<'s>` such that `Monoid.Gen(zero,add)` $\models$ Monoid.

```
module Monoid =
    let Gen(zero,add) =
        { new Monoid<_> with
            member this.zero = zero
            member this.add = fun x y ->
                match x,y with
                | z,a | a,z when z = zero -> a
                | _,_ -> add x y
        }
```

The additional pattern matches on the `add` method are activated when either of the parameters to the addition operator is the monoid's zero element, in which case the value conditions (criterion 5 in the above table) are programmatically enforced on the generated monoid `add` operand to ensure its specification is satisfied.

In this thesis we are only concerned about the correctness of the *generated* methods and modules. In the example above, we can show that the generated object actualizes the specifications for a monoid module (by the definitions in §4) if the provided program for `add` and the value provided for `zero` satisfy their respective specifications. It would be naïve to attempt (or even assume possible) to formally prove that a F# function for `add` satisfies the specification of Monoid.add as we have no formal framework or denotational semantics for such formal proof in F#. However, we can informally claim that a given code listing satisfies the given specification by simple analysis and examination of the program.

Implementation 5.12 (Field Generator) The field generator, very similar to the monoid generator, requires the field constants `zero` and `one` and the operators `neg`, `add`, `mul`, `inv`, `sub`, `div`,and `pow` to be supplied to the generator and satisfy their corresponding specifications. However, the methods `sub`, `div` and `pow` are optional parameters where the user may opt to pass no implementation (using the `None` value of the option type). When an implementation of any of these methods is omitted, a corresponding method is generated by using various techniques for the specification criteria defined earlier to satisfy the specification of the module members. For example, the `div` function may be generated using the value profiles defined in Interface 5.6 for division using the supplied `inv` and `mul` functions.

In this example we can see many of the specification criteria discussed earlier in the implementation of fields.

```
module Field =
    let Gen(zero,one,neg,add,mul,inv,sub,div,pow) =
        let div = fun x y ->
            match x,y,div with
            | a,z,_ when z = zero -> raise(DivideByZeroException())
            | z,a,_ when z = zero -> zero
            | a,o,_ when o = one -> a
            | o,a,_ when o = one -> inv a
            | a,b,_ when a = b -> one
            | x,y,None -> mul x (inv y)
            | x,y,Some d -> d x y
        let r =
            QuotientRing.Gen(
                zero,one,neg,add,mul,sub,pow,
                None,
                Some(fun x y -> Option.Some (div x y)),
                Some(fun _ _ -> zero),
                Some(fun _ _ -> one),
                Some mul,
                Some(fun _ x -> x))
        { new Field<_> with
            member this.zero = r.zero
            member this.one = r.one
            member this.neg = r.neg
            member this.add = r.add
            member this.mul = r.mul
            member this.sub = r.sub
            member this.div = div
            member this.div = r.div
            member this.rem = r.rem
            member this.gcd = r.gcd
            member this.lcm = r.lcm
            member this.τ = r.τ
            member this.longdiv = r.longdiv
            member f.inv = function
                | z when z = zero -> raise(DivideByZeroException())
                | o when o = one -> one
                | a -> inv a
            member f.pow = fun x y ->
                match pow with
                | Some p -> p x y
                | None ->
```

```
            let y = GetV y
            if y < 0 then f.inv(r.pow x (V -y))
            else r.pow x (V y)
    }
```

It may be worthwhile to analyze some common instances of the `Field` object to show how we can generate module instances that implement this module interface.

Recall the definition of `Value` type from Definition 3.14. Here, we will see the first instance of these value types in action, as each one of the operators contains both an implementation on "static" values, which are the expressions that are computed at calling time, and an implementation on "dynamic" values, which are the expressions that are computed at run time by evaluation[3].

Example 5.13 (Rationals Field Instance) Recall the definitions of `UnaryOp` from Definition 3.16, `BinaryOp` from Definition 3.17 and `V` as type constructor of `Value` objects from Definition 3.14. The object `QQ` defines a field of rational numbers using `BigRational` objects defined in .NET libraries, represented at a quotient of two `BigInteger` values.

```
let QQ =
    let zero = V 0N
    let one = V 1N
    let neg = UnaryOp.Gen (fun x -> - x) (fun x -> <@ - %x @>)
    let add = BinaryOp.Gen (fun x y -> x + y) (fun x y -> <@ %x + %y @>)
    let mul = BinaryOp.Gen (fun x y -> x * y) (fun x y -> <@ %x * %y @>)
    let inv = UnaryOp.Gen (fun x -> 1N / x) (fun x -> <@ 1N / %x @>)
    let sub = BinaryOp.Gen (fun x y -> x - y) (fun x y -> <@ %x - %y @>)
    let div = BinaryOp.Gen (fun x y -> x / y) (fun x y -> <@ %x / %y @>)
    let pow = BinaryOp.Gen (fun x y -> BigRational.PowN(x,y))
                           (fun x y -> <@ BigRational.PowN(%x,%y) @>)

    Field.Gen(zero, one, neg, add, mul, inv, Some sub, Some div, Some pow)
```

Notice that the generated object provides the implementation for subtraction, division, an exponentiation which are all optional. Instead of generating an implementation to these methods, we have provided the native operands which are much more efficient. The values provided for `zero` and `one` are only presented in the static form (using the `V` type constructor) as they can simply be resolved at compile time, but may also be lifted (trivially) to a dynamic expression if needed.

[3]Alternatively, these dynamic expression operators are often called transformers.

We claim that, assuming the implementation of rationals in .NET library is correct, then this object implements a field, i.e. `QQ` $\|\!=$ Field.
□

Similar to the example above, we may define an implementation of (pseudo-)Field `CC` for approximation of complex numbers $\mathbb{C}$ using the `Complex` library in .NET:

```
let CC = Field.Gen(V Complex.Zero, V Complex.One, ···)
```

We also define some similar implementations of QuotientRing module. Namely, `ZI` for machine integers — which have a well-known division with remainder algorithm — and `ZZ` for true integers using the `BigInteger` library.

Implementation 5.14 (Algebra Generator) The first step towards implementing an algebra module is to properly define the type (interface) of homomorphisms for the specific modules that are involved. Recall from Interface 5.9 that the `Homomorphism` object does not follow the conventional style of the other object types in this chapter and is instead parametrized in both the algebraic modules of the mapping and their respective underlying data sets. In this case we need to refine the definition of a homomorphism for each specific module type in this library. For example, the ring homomorphism type is defined as:

```
module Ring =
    type Homomorphism<'s,'t> =
         Homomorphism<'s,'t,Ring<'s>,Ring<'t>>
```

Similarly, an R-algebra homomorphism from R-algebra S to R-algebra T is defined as:

```
module Algebra =
    type Homomorphism<'r,'s,'t> =
         Homomorphism<'s,'t,Algebra<'r,'s>,Algebra<'r,'t>>
```

The generator for an `Algebra` object type requires only the ring homomorphism for the structural embedding of the coefficients ring to the elements rings. The scalar multiplication operator can then be automatically generated from the specifications from Interface 5.10. We will continue the definition of the **module** `Algebra` and add the following generator[4]:

[4]The vertical ellipses will always mean that the body of this **module** is being continued from an earlier definition.

```
module Algebra =
    ⋮
    let Gen(embed: Ring.Homomorphism<'s,'t>) =
        let coef = embed.s
        let elem = embed.d
        { new Algebra<_,_> with
            member this.zero = elem.zero
            member this.one = elem.one
            member this.neg = elem.neg
            member this.add = elem.add
            member this.mul = elem.mul
            member this.sub = elem.sub
            member this.pow = elem.pow
            member this.coefficients = coef
            member this.embedding = Homomorphism.Gen(coef,elem,embed.φ)
            member this.scalar = fun x y -> this.mul (embed.φ x) y
        }
```

Thus, we can infer that given a ring homomorphism $\varphi : S \to T$, such that $\varphi \models$ Homomorphism (Ring $S \to$ Ring T), then we can generate an R-algebra S such that `Algebra.Gen` $\varphi \models$ Algebra $R,\ S$.

Example 5.15 (Algebra Instances) A ring R corresponds to a trivial R-algebra R using the identity structural homomorphism, i.e. the scalar multiplication and ring multiplication are identical. This allows us to automatically lift any ring, quotient ring, or field to an algebra.

```
let GenAlgebraFromRing(r:Ring<_>) = Algebra.Gen(Homomorphism.Gen(r,r,id))
```

Such that, $r \models$ Ring $R \Rightarrow$ `GenAlgebraFromRing` $r \models$ Algebra $R,\ R$.
For example,

```
let ZZ = GenAlgebraFromRing QuotientRing.ZZ
let ZI = GenAlgebraFromRing QuotientRing.ZI
let QF = GenAlgebraFromRing Field.QF
let QQ = GenAlgebraFromRing Field.QQ
let CC = GenAlgebraFromRing Field.CC
```

□

5.1.5 Polynomial Algebras

We may now apply the same break-down of module interfaces and implementation techniques from §5.1.3 and §5.1.4 to define the object types for monomials and terms from §2.1.2 and polynomials from §2.1.4. Recall that monomials form a monoid, terms form a monomodule over these monomials, and polynomials form an algebra over the terms. Figure 5.3 defines the hierarchy of type refinements and inclusions among these objects and the other algebraic types defined earlier.

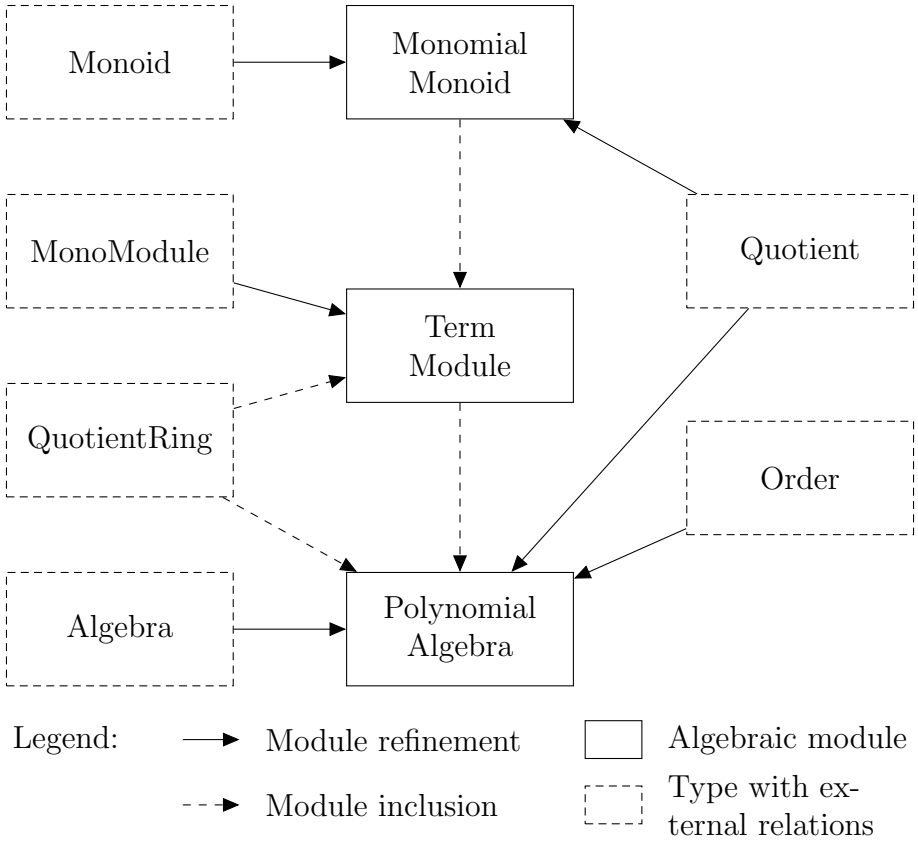

Figure 5.3: Interfaces and relations for polynomial objects

These objects form the three main feature modules that are used in the main algorithm for generation of program family members in this thesis. All these modules are parametrized over the other algebraic feature modules described in §5.1.3, but contain some very specific additional structure — such as monomials having a quotient structure with divisions, gcds, etc. but not quite forming a quotient ring — which we

define in this section.

Interface 5.16 (Monomial Monoid) As we showed in §2.1.2, monomials over a vector of variables $\vec{x} = [x_1, \ldots, x_n]$ form a monoid $\mathcal{M}$ over the set M with elements of the form $\prod_{i=1}^{n} x_i^{m_i}$ where $1_M = \prod_{i=1}^{n} x_i^{0_{\mathbb{N}}}$ and $\prod_{i=1}^{n} x_i^{m_i} \ *_M \ \prod_{i=1}^{n} x_i^{n_i} = \prod_{i=1}^{n} x_i^{m_i+n_i}$.

In the interface of the `Monoid` module, we chose to represent monoids additively, i.e. using 0_M and $+_M$ for the choice of the identity and the monoid operation. Monomials, however, are a natural occurrence of a *multiplicative* monoid. Here we will use a direct method of theory interpretation or transport [32] to define a 1_M constant and a $*_M$ binary operator that are defined to be equal to the `zero` and `add` values inherited from `Monoid` interface. This is a cosmetic change in code and does not mean that the monomial monoid has a ring form. We will take extra care to document this notation whenever it is used to resolve the confusion.

Additionally, monomials have gcd, lcm, crossfactors and divisibility operations as per Definition 2.13. The division operation here has no remainders, meaning that a monomial is either fully divisible by another monomial with zero remainder, or not divisible — this is due to the fact that monomials do not form a ring structure. We will implement these methods by overriding the `quotient` object defined earlier, but will refine the specifications to fully reflect this property.

The monomials interface also defines the access methods for obtaining the degree and logarithm of a monomial when requested, as well as an additional method `vars` which allows us to analyse every variable and its respective exponent that appears in the monomial. This allows us to maintain an abstract definition of a monomial while being able to access the crucial information in computing with monomials. Lastly, as an inverse transform to `log` and `vars`, every monomial monoid implementation must provide constructor functions that create a monomial from either a log value or an ordered set of variables and exponents.

Let n be the number of variables used in $\vec{x}$ for the monomials,

MonomialMonoid $M \ \sqsupseteq$ Monoid M:
  one: one = Monoid.zero
  $_{m,n}|\text{mul}|_p$: $m \in M \wedge n \in M \ \twoheadrightarrow$
    $\bar{p} = \text{Monoid.add}(m, n)$
  $_{m,n}|\text{div}|_d$: $m \in M \wedge n \in M \ \twoheadrightarrow$
    $\bar{d} \in M \cup \{\text{None}\} \ \wedge$
    $\bar{d} = \text{None} \Leftrightarrow n \not| m \ \wedge$
    $\bar{d} \neq \text{None} \Leftrightarrow m = \text{mul}(n, \bar{d}) \ \wedge$

$$
\begin{aligned}
&\quad n = \text{one} \Rightarrow \bar{d} = m \ \wedge \\
&\quad m = n \Rightarrow \bar{d} = \text{one} \\
&{}_{m,n}|\text{gcd}|_g : \ m \in M \wedge n \in M \ \rightarrowtriangle \\
&\quad \bar{g} \in M \ \wedge \\
&\quad \text{div}(m, \bar{g}) \neq \text{None} \wedge \text{div}(n, \bar{g}) \neq \text{None} \ \wedge \\
&\quad \forall_{g' \in M} \ \text{div}(m, g') \neq \text{None} \wedge \text{div}(n, g') \neq \text{None} \Rightarrow \text{div}(g, g') \neq \text{None} \\
&{}_{m,n}|\text{lcm}|_l : \ m \in M \wedge n \in M \ \rightarrowtriangle \\
&\quad \bar{l} \in M \ \wedge \\
&\quad \text{div}(\bar{l}, m) \neq \text{None} \wedge \text{div}(\bar{l}, n) \neq \text{None} \ \wedge \\
&\quad \forall_{l' \in M} \ \text{div}(l', m) \neq \text{None} \wedge \text{div}(l', n) \neq \text{None} \Rightarrow \text{div}(l', l) \neq \text{None} \\
&{}_{m,n}|\tau|_\tau : \ m \in M \wedge n \in M \ \rightarrowtriangle \\
&\quad \bar{\tau} \in M \ \wedge \\
&\quad \text{mul}(\bar{\tau}, m) = \text{lcm}(m, n) \ \wedge \\
&\quad \text{mul}(\bar{\tau}, \text{gcd}(m, n)) = n \ \wedge \\
&{}_{m}|\text{deg}|_d : \ m \in M \ \rightarrowtriangle \\
&\quad \bar{d} \in \mathbb{N} \ \wedge \\
&\quad \bar{d} = \mathit{deg}(m) \\
&{}_{m}|\text{log}|_l : \ m \in M \ \rightarrowtriangle \\
&\quad \bar{l} \in \mathbb{N}^n \ \wedge \\
&\quad \bar{l} = \mathit{log}(m) \\
&{}_{m}|\text{vars}|_v : \ m \in M \ \rightarrowtriangle \\
&\quad \bar{v} \in (\mathbb{N} \times \mathbb{N})^n \ \wedge \\
&\quad m = \textstyle\prod_{i, e \in \bar{v}} x_i^e \\
&{}_{l}|\text{fromLog}|_m : \ l \in \mathbb{N}^n \ \rightarrowtriangle \\
&\quad \bar{m} \in M \ \wedge \\
&\quad \text{log}(\bar{m}) = l \\
&{}_{v}|\text{fromVars}|_m : \ v \in (\mathbb{N} \times \mathbb{N})^n \ \rightarrowtriangle \\
&\quad \bar{m} \in M \ \wedge \\
&\quad \text{vars}(\bar{m}) = v
\end{aligned}
$$

```
type MonomialMonoid<'m> =
    inherit Monoid<'m>
    inherit Quotient<'m>

    abstract deg: UnaryOp<'m,int>
    abstract log: UnaryOp<'m,seq<int>>
    abstract vars: UnaryOp<'m,seq<int*int>>

    abstract fromLog: UnaryOp<seq<int>,'m>
    abstract fromVars: UnaryOp<seq<int*int>,'m>
```

Implementation 5.17 (Monomial Monoid Generator) The implementation of monomial monoids is similar to the other instance generators we encountered in §5.1.4. The following method generates an object which implements `MonomialMonoid`:

```
module MonomialMonoid =
    let Gen(one,mul,deg,log,vars,mkLog,mkVars,div,gcd,lcm,τ) =
        let m = Monoid.Gen(one,mul)
        let q = Quotient.Gen(one,None,None,Some div,None,Some gcd,Some lcm,Some τ)
        { new MonomialMonoid<_> with
            member this.zero = m.zero
            member this.add = m.add
            member this.deg = deg
            member this.log = log
            member this.vars = vars
            member this.fromLog = mkLog
            member this.fromVars = mkVars
            member this.longdiv = q.longdiv
            member this.div = q.div
            member this.rem = q.rem
            member this.gcd = q.gcd
            member this.lcm = q.lcm
            member this.τ = q.τ
        }
```

There are many ways of implementing monomials, but the two most common methods are dense and sparse representations.

Definition 5.18 (Dense Monomials) A *dense* monomial representation stores all the exponents of the variables $x_1, \ldots, x_n$ even if an exponent is zero. This is essentially equivalent to representing monomials by their logarithm and is useful when most of the variables are used in each monomial (i.e. when it is dense). In this case, the `add` method of the monomial monoid (monomial multiplication) is addition of the logarithm vectors: $\log(m * n) = \log(m) +_{\mathbb{N}^n} \log(n)$. Additionally, the degree of a monomial is simply the sum of all the exponents in the logarithm vector: $\deg(m) = \sum_{i \in \log(m)} i$. A dense monomial is implemented in the following way:

```
module MonomialMonoid =
    ⋮

    let LogToVars lg = seq {
```

```
    let c = ref 0
    for e in lg do
        if e > 0 then yield !c,e
        incr c
}

let VarsToLog vs = seq {
    let n = ref 0
    for i,e in vs do
        while !n < i do
            incr n
            yield 0
        incr n
        yield e
}

let SortVars vs = Seq.sortBy fst <| seq { for _,e as v:int*int in vs do if e > 0
     then yield v }

let DenseMap2 f a b =
    let rec loggen = function
    | l,[] | [],l -> l
    | e1::l1,e2::l2 -> (f e1 e2)::loggen(l1,l2)
    loggen(Seq.toList a, Seq.toList b) |> Seq.ofList

let Dense =
    let map2 f = TernaryOp.Lift DenseMap2 <@ DenseMap2 @> (BinaryOp.Flatten f)

    Gen(V Seq.empty,
        map2 Idx.add,
        UnaryOp.Lift Seq.sum <@ Seq.sum @>,
        id,
        UnaryOp.Lift LogToVars <@ LogToVars @>,
        id,
        UnaryOp.Gen (fun a -> VarsToLog(SortVars a))
                    (fun a -> <@ VarsToLog(SortVars %a) @>),
        (fun m n -> codegen {
            use d = map2 Idx.sub m n
            let c = Seq.Exists (UnaryOp.Flatten <| Bool.gt Idx.zero) d
            return Control.If c (V None) (Option.Some d)
        }) |> Fun2 |> Generate |> Function.Apply2,
        map2 Min,
```

```
        map2 Max,
        map2 (fun a b -> Max Idx.zero (Idx.sub b a)))
```

By code analysis, we claim that `MonomialMonoid.Dense` $\models$ MonomialMonoid.

Definition 5.19 (Sparse Monomials) In contrast to the dense implementation of monomials, the *sparse* representation only stores those variables whose exponents are non-zero. This is equivalent to storing the monomials as the `vars` list as a sequence of pairs of numbers, (i, e), where the first number i defines the variable x_i and the second number describes the exponent of this variable. Sparse representations are particularly useful when the number of variables in a polynomial algebra is large, but the size of each monomial is small in contrast. We implement the sparse monomials in the following way:

```
module MonomialMonoid =
    ⋮

    let SparseMap2 f a b =
        let rec vargen: _ -> (int*int)list = function
        | l,[] | [],l -> l
        | (i1,e1)::v1,(i2,e2)::v2 when i1 = i2 -> (i1,f e1 e2)::vargen(v1,v2)
        | (i1,_ as p1)::v1,((i2,_)::_ as n2) when i1 < i2 -> p1::vargen(v1,n2)
        | n1,p2::v2 -> p2::vargen(n1,v2)
        vargen(Seq.toList a, Seq.toList b) |> Seq.ofList

    let Sparse =
        let map2 f = TernaryOp.Lift SparseMap2 <@SparseMap2@> (BinaryOp.Flatten f)

        Gen(V Seq.empty,
            map2 Idx.add,
            UnaryOp.Gen (fun m -> Seq.sumBy snd m)
                        (fun m -> <@ Seq.sumBy snd %m @>),
            UnaryOp.Lift VarsToLog <@ VarsToLog @>,
            id,
            UnaryOp.Lift LogToVars <@ LogToVars @>,
            UnaryOp.Lift SortVars <@ SortVars @>,
            (fun m n -> codegen {
                use d = map2 Idx.sub m n
                let f = Tuple.Snd >> Bool.gt Idx.zero
                let c = Seq.Exists (UnaryOp.Flatten f) d
                return Control.If c (V None) (Option.Some d)
```

```
}) |> Fun2 |> Generate |> Function.Apply2,
map2 Min,
map2 Max,
map2 (fun a b -> Max Idx.zero (Idx.sub b a)))
```

Similarly, we claim that `MonomialMonoid.Sparse` $\Vdash$ MonomialMonoid.

Note that since the arrays and lists in F# are 0-indexed, we refer to the variables in $\vec{x}$ by $x_0, \ldots, x_{n-1}$ instead of $x_1, \ldots, x_n$.

Combining the monomial monoid with a ring of coefficients leads to a natural monomodule structure for the terms:

Interface 5.20 (Term MonoModule) The term monomodule is a parametrized feature module which depends on an external ring of coefficients C, implemented as a `QuotientRing` (to support term division) and a monoid of monomials M as the elements of the base monoid. The type of the terms T is left as an open abstract type to be decided by the implementation.

The term monomodule interface contains accessor methods for retrieving either the coefficient or the monomial of a term, and a constructor method to create a term in T from a coefficient in C and a monomial in M:

TermModule C, M, T $\sqsupseteq$ MonoModule C, T:
- CR: CR $\Vdash$ QuotientRing C
- MM: MM $\Vdash$ MonomialMonoid M
- one: one = Monoid.zero
- ${}_{t,s}|\text{mul}|_p$: $t \in T \wedge s \in T \;\rightharpoonup$
  - $\bar{p} = \text{Monoid.add}(t, s)$
- ${}_{t,s}|\text{div}|_d$: $t \in T \wedge s \in T \;\rightharpoonup$
  - $\bar{d} \in T \cup \{\text{None}\} \;\wedge$
  - $\bar{d} = \text{None} \Leftrightarrow s \not| t \;\wedge$
  - $\bar{d} \neq \text{None} \Leftrightarrow t = \text{mul}(s, \bar{d}) \;\wedge$
  - $s = \text{one} \Rightarrow \bar{d} = t \;\wedge$
  - $t = s \Rightarrow \bar{d} = \text{one}$
- ${}_{t}|\text{c}|_c$: $t \in T \;\rightharpoonup$
  - $\bar{c} \in C \;\wedge$
  - $\forall_{s \in T}\; \text{CR.mul}(\bar{c}, \text{c}(s)) = \text{c}(\text{mul}(t, s))$
- ${}_{t}|\text{m}|_m$: $t \in T \;\rightharpoonup$
  - $\bar{m} \in M \;\wedge$
  - $\forall_{s \in T}\; \text{MM.mul}(\bar{m}, \text{m}(s)) = \text{m}(\text{mul}(t, s))$
- ${}_{c,m}|\text{make}|_t$: $c \in C \wedge m \in M \;\rightharpoonup$
  - $\bar{t} \in T \;\wedge$

$$\mathrm{c}(\bar{t}) = c \ \wedge \ \mathrm{m}(\bar{t}) = m$$

```
type TermModule<'c,'m,'t> =
    inherit MonoModule<'c,'t>

    abstract CR: QuotientRing<'c>
    abstract MM: MonomialMonoid<'m>

    abstract div: BinaryOp<'t,'t,'t option>

    abstract c: UnaryOp<'t,'c>
    abstract m: UnaryOp<'t,'m>
    abstract make: BinaryOp<'c,'m,'t>
```

Implementation 5.21 (Term Generator) Unlike the previous implementations, the generator for term monomodule requires a quotient ring of coefficients to already be defined. This generator — similar to the `Algebra` generator — defines a feature module which is parametrized by other feature modules.

```
module TermModule =
    let Gen(cr:QuotientRing<_>,mm,one,mul,div,scalar,getc,getm,make) =
        let e = Monoid.Gen(one,mul)
        let m = MonoModule.Gen(e,cr,scalar)
        { new TermModule<_,_,_> with
            member this.zero = m.zero
            member this.add = m.add
            member this.scalar = m.scalar
            member this.coefficients = cr :> Ring<_>
            member this.CR = cr
            member this.MM = mm
            member this.div = div
            member this.c = getc
            member this.m = getm
            member this.make = make
        }
```

There are many possible implementations of terms which we will cover for each specific instance of the problem. For example, if we know that the problem requires only polynomials of degree one (in the case of linear polynomials which define a Gaussian Elimination problem), then we know that each term contains at most only one variable with multiplicity one and can specialize the implementation of the terms

to reflect that property. The following example shows the most generic purpose implementation of the terms:

Example 5.22 (Generic Terms) The most generic form of term is implemented as a pair of a coefficient and a monomial, where the user specifies the choice of a quotient ring for the coefficients (called `cr` in this code) and a monomial monoid for the monomials (called `mm`) and the term accessor methods are simply projections on these underlying types.

```
module TermModule =
    ⋮
    let Generic(cr,mm) =
        Gen(cr, mm,
            Pair cr.one mm.zero,
            (fun a b -> Pair (cr.mul (Fst a) (Fst b)) (mm.add (Snd a) (Snd b))),
            (fun a b -> Generate <| codegen {
                use c = cr.div (Fst a) (Fst b)
                use m = mm.div (Snd a) (Snd b)
                return! If ((IsSome c) ^&& (IsSome m))
                          (Return <| Some(Pair (UnOption c) (UnOption m)))
                          (Return <| Option None)
            }),
            (fun s t -> Pair (cr.mul s (Fst t)) (Snd t)),
            Fst, Snd, Pair)
```

□

Finally, the ultimate feature module in the polynomial algebra library is the specialization of the `Algebra` module which specifies and implements the polynomial algebra as per Definition 2.22.

Interface 5.23 (Polynomial Algebra) The polynomial algebra is a parametrized feature module that depends on an external ring of coefficients C and a term module T which uses these coefficients. Additionally, a monomial monoid M is also required due to the dependency of the term monomodule. The final type of polynomials P is defined as an open type to be implemented by individual specialization of polynomial algebra.

The polynomials P are treated as a generic collection of terms, where the only accessors for the terms are the methods `LT` which returns the leading term of the

polynomial according to the term ordering defined by the polynomial algebra, and `RT` which returns the polynomial with its leading term removed. The method `RT` is essentially equivalent to lifting the leading term as a polynomial and subtracting it from the original value but is provided as a separate method in order to allow optimizations in the implementation. Similarly, the methods `LM` and `LC` access the leading monomial and coefficient of the polynomial respectively.

In addition to refining the `Algebra` feature module, the polynomial algebra module also refines the `Quotient` aspect module to define the division and remainder operations among polynomials. Additionally, polynomial algebra also refines the `Order` feature module on the monomials which defined the term ordering as defined in §2.1.3.

PolynomialAlgebra $C,\ M,\ T,\ P \sqsupseteq$ Algebra $C,\ P \cup$ Order M:

CR: CR $\Vdash$ QuotientRing C

TM: TM $\Vdash$ TermModule $C,\ M,\ T$

${}_{p,q}|\text{longdiv}|_{d,r}$: $p \in P \wedge q \in P \ \rightarrowtriangle$

$\bar{d} \in P \wedge \bar{r} \in P \ \wedge$

$p = \text{add}(\text{mul}(q, \bar{d}), \bar{r}) \ \wedge$

$\text{longdiv}(\bar{r}, q) = \text{zero}, \bar{r} \ \wedge$

$p = \text{zero} \Rightarrow \bar{d} = \text{zero} \wedge \bar{r} = \text{zero} \ \wedge$

$q = \text{zero} \Rightarrow \bar{d} = \text{zero} \wedge \bar{r} = p \ \wedge$

$q = \text{one} \Rightarrow \bar{d} = p \wedge \bar{r} = \text{zero} \ \wedge$

$p = q \Rightarrow \bar{d} = \text{one} \wedge \bar{r} = \text{zero}$

${}_{p,q}|\text{div}|_{d}$: $p \in P \wedge q \in P \ \rightarrowtriangle$

$\bar{d} \in P \cup \{\text{None}\} \ \wedge$

$\bar{d} \neq \text{None} \Leftrightarrow \text{longdiv}(p, q) = \bar{d}, \text{zero}$

${}_{p,q}|\text{rem}|_{r}$: $p \in P \wedge q \in P \ \rightarrowtriangle$

$\bar{r} \in P \ \wedge$

$\exists_{d \in S} \ \text{longdiv}(p, q) = d, \bar{r}$

${}_{p}|\text{terms}|_{\vec{t}}$: $\bar{p} \in P \ \rightarrowtriangle$

$\vec{t} \subset T \ \wedge$

$\text{supp}(p) = \vec{t}$

${}_{p}|\text{deg}|_{d}$: $\bar{p} \in P \ \rightarrowtriangle$

$\bar{d} \in \mathbb{N} \ \wedge$

$\text{deg}(p) = \bar{d}$

${}_{p}|\text{isZero}|_{z}$: $\bar{p} \in P \ \rightarrowtriangle$

$\bar{z} \in \mathbb{B} \ \wedge$

$\bar{z} \Leftrightarrow p = \text{zero}$

${}_{p}|\text{LT}|_{t}$: $p \in P \ \rightarrowtriangle$

$\bar{t} \in T \ \wedge$

$\bar{t} \in \text{supp}(p) \ \wedge$

$\forall_{t' \in \text{supp}(p)} \ \text{cmp}(\text{TM.m}(t), \text{TM.m}(t')) > 0$

$_p|\mathrm{LM}|_m$: $p \in P \;\; \rightarrowtriangle$
$\bar{m} \in M \;\wedge$
$\bar{m} = \mathrm{TM.m}(\mathrm{LT}(p))$
$_p|\mathrm{LC}|_c$: $p \in P \;\; \rightarrowtriangle$
$\bar{c} \in C \;\wedge$
$\bar{c} = \mathrm{TM.c}(\mathrm{LT}(p))$
$_p|\mathrm{RT}|_q$: $p \in P \;\; \rightarrowtriangle$
$\bar{q} \in P \;\wedge$
$\mathrm{supp}(\bar{q}) = \mathrm{supp}(p) \backslash \{\mathrm{LT}(p)\}$
$_{\vec{t}}|\mathrm{formTerms}|_p$: $\vec{t} \subset T \;\; \rightarrowtriangle$
$\bar{p} \in P \;\wedge$
$\mathrm{supp}(\bar{p}) = \vec{t}$
$_{c,m}|\mathrm{FromTerm}|_p$: $c \in C \wedge m \in M \;\; \rightarrowtriangle$
$\bar{p} \in P \;\wedge$
$\mathrm{supp}(\bar{p}) = \{\mathrm{TM.make}(c, m)\}$

```
type PolynomialAlgebra<'c,'m,'t,'p> =
    inherit Order<'m>
    inherit Algebra<'c,'p>
    inherit Quotient<'p>

    abstract CR: QuotientRing<'c>
    abstract TM: TermModule<'c,'m,'t>

    abstract terms: UnaryOp<'p,seq<'t>>
    abstract deg: UnaryOp<'p,int>
    abstract isZero: UnaryOp<'p,bool>

    abstract LT: UnaryOp<'p,'t>
    abstract LM: UnaryOp<'p,'m>
    abstract LC: UnaryOp<'p,'c>
    abstract RT: UnaryOp<'p,'p>

    abstract fromTerm: BinaryOp<'c,'m,'p>
    abstract fromTerms: UnaryOp<seq<'t>,'p>
```

Implementation 5.24 (Polynomial Algebra Generator) The polynomial algebra generator requires a quotient ring of coefficients, a term monomodule, and a polynomial ring to generate an algebra of polynomials as described by Interface 5.23. Figure 5.4 shows the parameters used by this generator and their type dependencies.

If a degree-compatible term ordering is provided and the `deg` parameter is not

supplied, a degree operation is automatically generated which returns the degree of the leading term. Similarly, if the LM and LC operators are not supplied, a standard implementation is generated which returns the projections of the leading term.

```
module PolynomialAlgebra =
    let Gen(cr:QuotientRing<_>,tm,pr:QuotientRing<_>,
            scalar,cmp,terms,deg,iszero,lt,lm,lc,rt,make,mkterm) =
        let a = Algebra.Gen(Homomorphism.Gen(cr:>Ring<_>,pr:>Ring<_>,scalar))
        { new PolynomialAlgebra<_,_,_,_> with
            member this.zero = a.zero
            member this.one = a.one
            member this.neg = a.neg
            member this.add = a.add
            member this.mul = a.mul
            member this.sub = a.sub
            member this.pow = a.pow
            member this.coefficients = a.coefficients
            member this.embedding = a.embedding
            member this.scalar = a.scalar
            member this.cmp = cmp
            member this.CR = cr
            member this.TM = tm
            member this.terms = terms
            member this.deg = match deg with
                              | Some deg -> deg
                              | _ -> fun p -> tm.MM.deg(tm.m(lt p))
            member this.isZero = iszero
            member this.LT = lt
            member this.RT = rt
            member this.LM = match lm with
                              | Some lm -> lm
                              | _ -> fun p -> tm.m(lt p)
            member this.LC = match lc with
                              | Some lc -> lc
                              | _ -> fun p -> tm.c(lt p)
            member this.fromTerms = make
            member this.fromTerm = mkterm
            member this.longdiv = pr.longdiv
            member this.div = pr.div
            member this.rem = pr.rem
            member this.gcd = pr.gcd
            member this.lcm = pr.lcm
```

Name	Type	Description
`cr`	`QuotientRing<'C>`	The ring of coefficients C for polynomial algebra. This should normally be the same object as the coefficients of the term monomodule.
`tm`	`TermModule<'C,'M,'T>`	The term monomodule T of the terms used in the polynomials.
`pr`	`QuotientRing<'P>`	The polynomial ring P defines the internal operations and the division algorithm on the polynomials.
`scalar`	`UnaryOp<'C,'P>`	The scalar lifting operator defines how a scalar C is embedded as a constant polynomial P.
`cmp`	`BinaryOp<'M,'M,int>`	The ordering operation between the monomials of type M implements the `Order` interface for polynomial algebra.
`terms`	`UnaryOp<'P,seq<'T>>`	The projection operator from a polynomial P to its support as a set of terms T.
`deg`	`UnaryOp<'P,int>`	*(optional)* The unary operator `deg` returns the degree of the polynomial P. A standard implementation is generated if this value is `None`.
`isZero`	`UnaryOp<'P,bool>`	Provides a shorthand for code generators for quickly test a polynomial against the zero polynomial.
`lt`	`UnaryOp<'P,'T>`	The `lt` operator returns the leading term of the polynomial. Must be compatible with the monomial ordering provided.
`lm`	`UnaryOp<'P,'M>`	*(optional)* Returns the leading monomial of the polynomial.
`lc`	`UnaryOp<'P,'C>`	*(optional)* Returns the leading coefficient of the polynomial.
`rt`	`UnaryOp<'P,'P>`	Removes the leading term from a polynomial and returns the remaining terms as a new polynomial.
`fromTerms`	`UnaryOp<seq<'T>,'P>`	Creates a polynomial object P from a set of terms in T.
`fromTerm`	`BinaryOp<'C,'M,'P>`	Creates a polynomial P as the lifting of a term. This method must be equivalent to the `make` constructor method.

Figure 5.4: Parameters of the Polynomial Algebra Generator

```
        member this.τ = pr.τ
    }
```

The `PolynomialAlgebra` feature module is the basis of the algebra library in this thesis and provides specialized routines for all the operations on polynomials. The specializations of this module define the most important optimizations that may be performed on specific problems. For example, a specialized implementation of polynomial algebra for linear (degree one) polynomials is what defines a problem of Gaussian Elimination instead of a generalized computation of a Gröbner basis. Similarly, a special implementation of univariate polynomials is the feature that defines the Euclidean algorithm for polynomials GCDs.

Before showing an example of a generic polynomial algebra, we first need to show some implementations of term ordering that we may use.

Example 5.25 (Term Ordering for Dense Monomials) Implementation of the term orderings is highly dependent on the choice of monomials used. In this example we will cover the four standard term orderings (`Lex`, `RevLex`, `DegLex` and `DegRevLex`) for dense monomials as implemented in Definition 5.18.

The following two methods show an implementation of lexicographical term ordering and reverse lexicographical ordering given the logarithms of two monomials:

```
let rec lex = function
    | [],[] -> 0
    | _,[] -> 1
    | [], _ -> -1
    | e1::l1,e2::l2 ->
        match compare e1 e2 with
        | 0 -> lex (l1,l2)
        | x -> x

let rec revlex o = function
    | [],[] -> o
    | _,[] -> -1
    | [], _ -> 1
    | e1::l1,e2::l2 ->
        match compare e1 e2 with
        | 0 -> revlex o (l1,l2)
        | x -> revlex -x (l1,l2)
```

Using these methods, we may simply implement the aforementioned term orderings as:

```
module TermOrder =
    let Lex (m:#MonomialMonoid<_>) =
        let cmp (a:seq<int>) b = lex (Seq.toList a, Seq.toList b)
        Order.Gen (fun a b -> Function.Apply2 (E<@cmp@>) (m.log a) (m.log b))
    let RevLex (m:#MonomialMonoid<_>) =
        let cmp (a:seq<int>) b = revlex 0 (Seq.toList a, Seq.toList b)
        Order.Gen (fun a b -> Function.Apply2 (E<@cmp@>) (m.log a) (m.log b))
    let DegLex (m:#MonomialMonoid<_>) =
        let cmp (d1:int,m1:seq<int>) (d2,m2) =
            match compare d1 d2 with
            | 0 -> lex (Seq.toList m1, Seq.toList m2)
            | n -> n
        Order.Gen (fun a b -> Function.Apply2 (E<@cmp@>)
                              (Tuple.Pair (m.deg a) (m.log a))
                              (Tuple.Pair (m.deg b) (m.log b)))
    let DegRevLex (m:#MonomialMonoid<_>) =
        let cmp (d1:int,m1:seq<int>) (d2,m2) =
            match compare d1 d2 with
            | 0 -> revlex 0 (Seq.toList m1, Seq.toList m2)
            | n -> n
        Order.Gen (fun a b -> Function.Apply2 (E<@cmp@>)
                              (Tuple.Pair (m.deg a) (m.log a))
                              (Tuple.Pair (m.deg b) (m.log b)))
```

Notice that if the monomial monoid provided to these order generating functions is a sparse implementation, the resulting `Order` object is still correct and functional, but it is not optimal as each comparison method forces the monomials to be converted to the logarithm representation before comparison. A similar implementation may be provided for operating on the variables list of sparse monomials instead.
□

Example 5.26 (Generic Polynomials) We may now implement a generic version of the polynomial algebra using the terms defined in Example 5.22. Provided a term monomodule (such as the ones generated by `TermModule.Generic`) and a term ordering on compatible monomials (such as those generated in Example 5.25), we generate a generic term algebra as a list of terms as follows:

```
module PolynomialAlgebra =
```

```
⋮
let Generic(tm:TermModule<_,_,_>,o:Order<_>) =
    let app0,app1,app2 = GetV, UnaryOp.AppV, BinaryOp.AppV
    let getm,getc = app1 tm.m, app1 tm.c
    let sort = List.sortWith (fun a b -> - app2 o.cmp (getm a) (getm b))
    let rec collapse = function
        | [] -> []
        | x::y::ts when app2 o.cmp (getm x) (getm y) = 0 -> app2 tm.make (app2
            tm.CR.add (getc x) (getc y)) (getm x) :: ts |> collapse
        | x::ts when getc x = app0 tm.CR.zero -> collapse ts
        | x::ts -> x :: collapse ts
    let coefmap f = List.map (fun t -> app2 tm.make (app1 f (getc t)) (getm t))
    let scalar s = [app2 tm.make s (app0 tm.MM.zero)] |> collapse
    let neg p = coefmap (tm.CR.neg) p
    let add p g = p @ g |> sort |> collapse
    let sub p g = add p (neg g)
    let mul (p:_ list) (g:_ list) = [for t in p do yield! [for s in g do yield
        app2 tm.add t s]] |> sort |> collapse
    let longdiv p g =
        let rec div x y =
            match x,y with
            | t,[] -> [],t
            | [],_ -> [],[]
            | m::ts,n::ss ->
                match app2 tm.div m n with
                | None -> [],x
                | Some q ->
                    let d,r = div (sub ts (mul ss [q])) y
                    q::d,r
        let d,r = div p g
        d |> sort |> collapse, r
    let pr =
        QuotientRing.Gen(
            scalar (app0 tm.CR.zero) |> V,
            scalar (app0 tm.CR.one) |> V,
            UnaryOp.Lift neg <@neg@>,
            BinaryOp.Lift add <@add@>,
            BinaryOp.Lift mul <@mul@>,
            BinaryOp.Lift sub <@sub@> |> Some,
            None,
            BinaryOp.Lift longdiv <@longdiv@> |> Some,
```

```
        None, None, None, None, None)
Gen(tm.CR, tm, pr,
    UnaryOp.Lift scalar <@scalar@>,
    None,
    o.cmp,
    List.ToSeq,
    Some (fun p -> Control.If (List.IsEmpty p) Idx.zero
                              (tm.MM.deg (tm.m (List.Head p)))),
    List.IsEmpty,
    List.Head,
    None, None,
    List.Tail,
    UnaryOp.Gen (fun ts -> collapse(sort(Seq.toList ts)))
                (fun ts -> <@ collapse(sort(Seq.toList %ts)) @>),
    fun c m -> tm.make c m |> List.Singleton)
```

□

5.2 Modular Decomposition of Buchberger's Algorithm

Recall from §4.2.4 that we provide the members of a program family by decomposing the architecture of the family into orthogonal and functional aspect modules and provide different implementations and connections between these modules according to the specializations available. In Example 4.28 we showed a sample decomposition of the KWIC software architecture following the criteria for this decomposition. We are now going to analyse and decompose Buchberger's algorithm in this section to provide a similar decomposition to be used in our code generator.

During this decomposition, we will break down Buchberger's algorithm in two different parts:

(1) The main algorithm: This is the main Buchberger's algorithm that computes Gröbner bases as per Algorithm 2.56 and further improved in Algorithm 2.61. The modules in this part correspond exactly to the computations performed by Buchberger's algorithm.

(2) The post processing algorithms: These parts are not included in Buchberger's algorithm originally, but are required to compute minimal Gröbner bases as per

Algorithm 2.57 and reduced Gröbner bases as per Algorithm 2.58. The modules in this part of the breakdown perform all the operations required to compute these derivatives of Gröbner bases.

5.2.1 Main Algorithm

The main part of Buchebrger's algorithm is decomposed into two feature modules and three distinct aspect modules:

(1) **Container**: This feature module represents a (polynomial) container for the current basis elements. The container module provides methods for adding new polynomials and retrieving them — either one at a time given a path index or the entire basis as a set of polynomials.

(2) **Working Set**: This feature module is responsible for storage and retrieval of critical pairs of polynomials to compute the S-polynomial. Buchberger's algorithm originally contained a straight-forward implementation of this module to analyse every pair of polynomials, but the two criteria in Lemmas 2.59 and 2.60 may be implemented as part of this selection strategy to provide better optimization. This module provides methods for adding, removing, and querying for critical pairs.

(3) **S-Polynomial**: This aspect module computes the S-polynomial of two given polynomials. This module may simply be implemented as defined in Definition 2.51 using the operations in the supplied polynomial algebra, but some instances of the problem may specialize to much simpler algorithms for this module.

(4) **Normal Remainder**: This aspect module is responsible for computing the normal remainder (residual) of a polynomial against the current basis members. If the residual of the S-polynomial from a selected pair is non-zero, then a new basis element is to be added. We have already seen two methods of implementing this module: either as remainder of the repeated division by basis elements (as defined in Definition 2.53) or as the normal form from a polynomial rewrite system as defined in §2.2.4. Theorem 2.39 showed that these two specializations are equivalent.

(5) **Expansion Strategy**: This aspect module works complementary to the selection strategy to expand the working set of critical pairs to be considered. A direct implementation of this module would add new critical pairs every time a new basis element is added to the Gröbner basis; however, some implementations may consider other expansions such as considerations of Buchberger triples as defined in Lemma 2.60 to the working set.

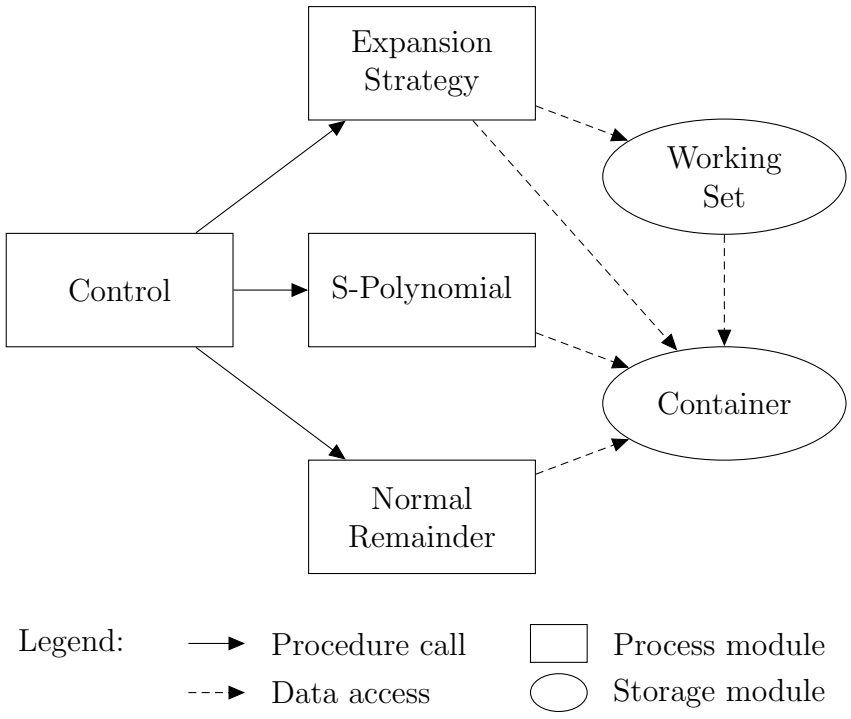

Figure 5.5: Architecture of main part of Buchberger's algorithm

Finally, we are introducing a control module at the top layer which controls the flow of data between the earlier modules. The relationship between these modules and the access and data flow are depicted in Figure 5.5.

It should come at no surprise that the container and working set feature modules are parametrized in the choice of polynomial algebra, and that the three processing aspect modules and the controller aspect module are parametrized by both the choice of the polynomial algebra and the choice of the container.

5.2.2 Post Processing

Similar to the main section of Buchberger's algorithm, the post processing component of the algorithm is decomposed into individual aspect modules for each of the actions. The principal difference between the aspect modules defined in here and those in §5.2.1 is that the post processing modules have the option of not being provided at all, and this translates to the choice of having a standard, a minimal, or a reduced Gröbner basis. The algorithm is broken down into the following aspect modules:

- **Reduction**: This module is responsible for the removal of superfluous polynomials in the computation of minimal Gröbner bases. Note that even though this is *reducing* the size of the basis, the result of a full implementation for this module leads to a *minimal* basis (not a reduced Gröbner basis). This module could be skipped by the controller if computation of the minimal basis is not required, but it could also be implemented by a divisibility test of leading terms as described by Definition 2.45 and Algorithm 2.57, or by other means of computing superfluous polynomials in combination with other modules such as the method described by Passmore and Moura [69]. This module provides a method for deciding whether a polynomial is reducible or not.

- **Canonicalization**: This module computes the canonical form of each polynomials in a minimal Gröbner basis to compute the reduced Gröbner basis for the input set of polynomials. Similar to the reduction module, the controller may choose not to pass data to this module when computation of the reduced basis is not required. When implemented, this module may produce the same canonical forms as defined in Definition 2.46 and Algorithm 2.58 for the choice of the polynomial algebra. Moreover, the implementer might choose a different specialization of this module with respect to a different canonicalization method, such as only computing monic polynomials (this is, in fact, equivalent to producing a row-echelon matrix when linear polynomials are involved). This module provides a method to canonicalize a basis element.

The post processing architecture shares the same polynomial container feature module as that used in the main algorithm, since it needs to perform computations on the same set of polynomials. The controller module for the post processing may choose which post processing operations are to be executed, and whether the same

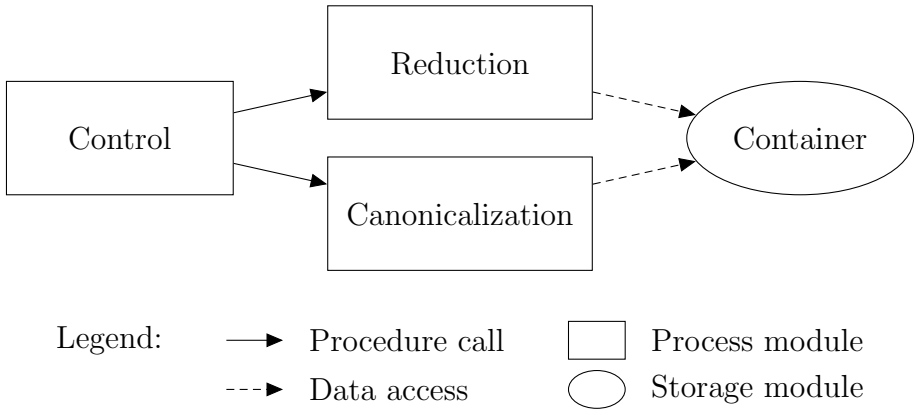

Figure 5.6: Post-processing architecture for reduced and minimal Gröbner bases

container should be used to output the result of the computation or a new container is to be created. Figure 5.6 demonstrates this decomposition.

Similar to the main algorithm, the two processing modules defined here are both parametrized by the choices for the polynomial algebra and the container of the polynomials.

5.2.3 Full Decomposition

Finally, we may combine the two architectures from the main processing section and the post processing section to provide a full decomposition of the software used in this thesis. This decomposition contains both the modules from Buchberger's algorithm and the related modules for processing the data to produce different variations and flavours of minimal and reduced bases. Additionally, this combined architecture unifies the polynomial container module used in both part of the algorithm and introduces a new controller module which directs the execution and the data flow between the multiple involved algorithms. Figure 5.7 provides the full decomposition of the software used in this thesis. This provides the modular design for the code generator that we program in the next section.

This architecture introduces two new aspect modules that the main controller uses for the input set of parameters from the user and the output basis result. The specialization provided by these modules defines the link between the statement of the original problem and the translation of this problem to a Gröbner basis computation problem:

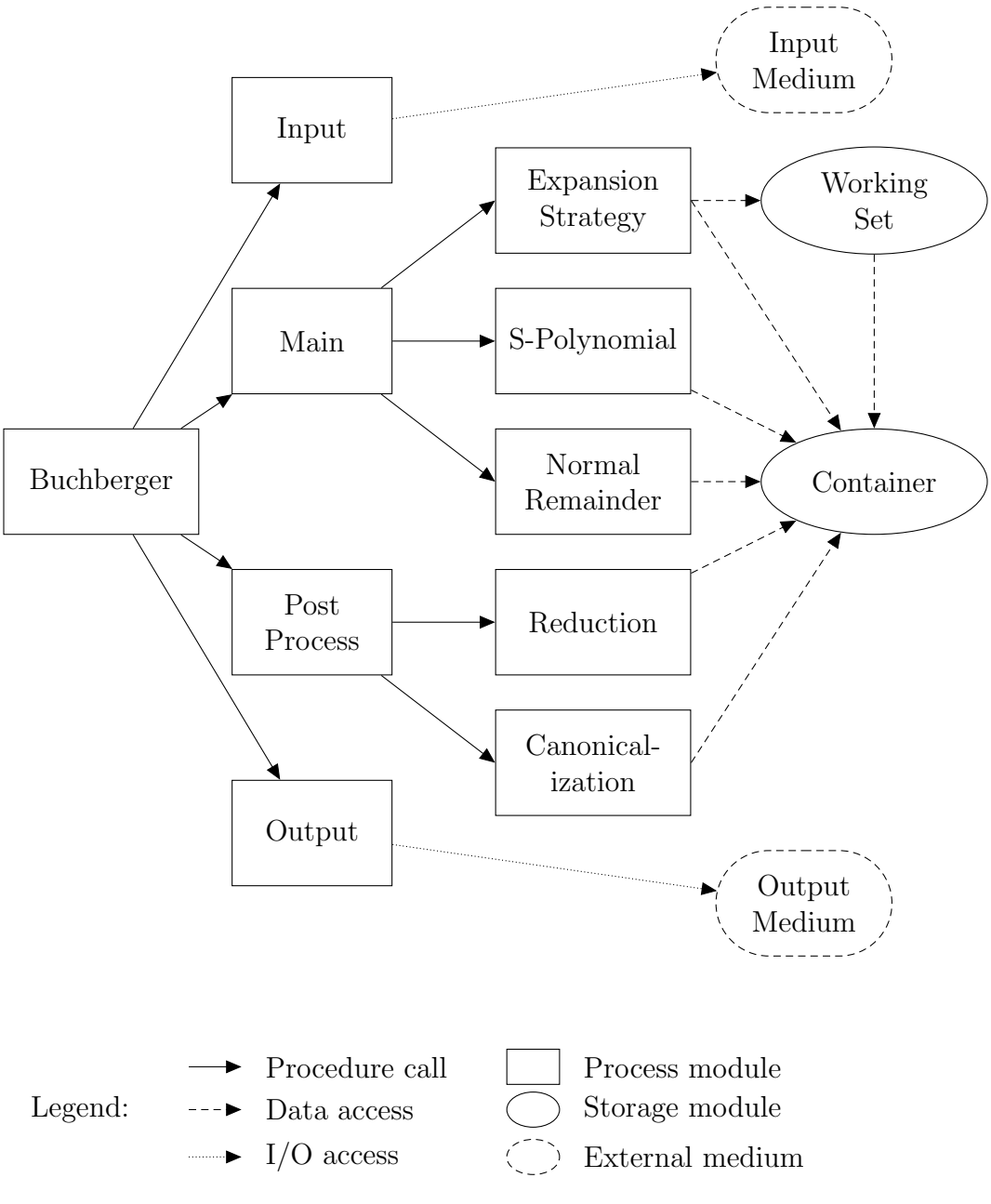

Figure 5.7: General architecture of Buchberger's algorithm

- **Input**: The input module is responsible for transforming the problem from the input domain into a set of polynomials to be solved using the computation of the Gröbner bases. In §2.4 we provided some of the most popular problems that can be solved using Gröbner bases and the methods for transforming the

statement of the problem to a question of ideal membership within some polynomial ideal and then interpreting the result as the solution to the proposed problem. This module performs the computation necessary for the first part of this transformation.

- **Output**: As a counterpart to the Input module, this module is responsible for the second part of the transformation where the result of the algorithm — provided as a set of polynomials for the Gröbner basis — is translated back into the domain of the problem to provide the answer.

5.3 Generative Version of Buchberger's Algorithm

We now have everything that we need for implementing the code generator to instantiate specializations of Buchberger's algorithm. We have the algebra framework in F# to use and compute with polynomial algebras, we have a code generation DSL to program the generator for this algorithm, and we have the modular breakdown for this program with the theoretical requirements to specialize them. In this section, we will finally assemble all the topics discussed in this thesis to program the code generator and show some examples of it.

5.3.1 Interfaces for Modules

We will first define the interfaces for all the modules introduced in §5.2. This includes the module for polynomial containers, and the aspect modules for input, output, selection strategy, expansion strategy, S-polynomial, normal remainder, reduction strategy, and canonical form operations. We do not have the interface for the control module, and the next section will program the control module as a part of the code generator.

These aspect modules are closely related to the idea of separation of concerns and abstract a programmatic section of the code in each method. The methods provided by these modules are either code generators or state operators. We will refrain from giving a formal mathematical interface for these modules at this point.

Interface 5.27 (Input Module) The `Input` module requires an open type `'inp` that defines the type of the input to the problem. The task of this module is to convert the input of the problem to a set of polynomials — given as the generic type `'itm`.

The `Process` method initializes the `Input` module and converts the input parameter to a sequence of polynomials to return to the main controller.

```
type Input<'itm,'inp> =
    abstract Process: Value<'inp> ->
            CodeGen<seq<'itm>,'w>
```

Interface 5.28 (Output Module) Similar to the `Input` module, the `Output` module is responsible to converting the computed Gröbner basis — given as a set of polynomials of generic type `'itm` — to the final output type of the problem which has the type `'out`.

The `Process` method initializes the `Output` module with the computed Gröbner basis and converts it to the output data type.

```
type Output<'itm,'out> =
    abstract Process: Value<seq<'itm>> ->
            CodeGen<'out,'w>
```

Interface 5.29 (Container Module) The `Container` module provides all the standard container access methods to maintain a set of polynomials. The open type `'cnt` is the type of the container object used. Some examples of container types that may be used are arrays, lists (vectors), or database access objects.

Similar to the earlier usage in `Input` and `Output` modules, the type `'itm` designates the type of polynomials used. Furthermore, an index access type `'idx` for polynomials in the container is provided for referring to the elements in this container. In the case of standard in-memory data structures such as lists and arrays, this "index" could simply be the polynomial itself, or in more complex implementations the index is the identifier for accessing the items from an external storage.

We have not included the removal operations in the container in this interface. This is due to the fact that the code generator in the next section does not delete any polynomials from the container, and thus we have not specified the removal operations to make the container type in the most generalized form.

```
type Container<'itm,'idx,'cnt> =
    abstract Init: Value<seq<'itm>> ->
            CodeGen<'cnt,'w>
    abstract Add: Value<'cnt> -> Value<'itm> ->
            CodeGen<'idx,'w>
    abstract Get: Value<'cnt> -> Value<'idx> ->
```

```
        CodeGen<'itm,'w>
    abstract All: Value<'cnt> ->
        CodeGen<seq<'itm>,'w>
    abstract Indexes: Value<'cnt> ->
        CodeGen<seq<'idx>,'w>
```

Interface 5.30 (Working Set Module) The `WorkingSet` is provided as the open type `'ws` with parameters `'idx` as the index of polynomials within the container and the container type `'cnt`. The methods in this module perform their respective operations when provided with both the working set and the basis container.

This module provides a code generator `Init` for initializing the working set of polynomial pairs and selection strategy, a code generator `HasMore` which determines if there are more critical pairs to be computed, a code generator `Pick` to generate (or retrieve) a new critical pair (as a tuple of indices) to compute, the `Add` and `Del` code generators to add or remove new pairs to or from the working set of critical pairs, and finally a code generator `All` that outputs the sequence of all remaining critical pairs for analysis purposes.

Note that the working set is essentially a specialization of a container with additional functionality for altering the storage method depending on the chosen selection strategy.

```
type WorkingSet<'idx,'cnt,'ws> =
    abstract Init: Value<'cnt> ->
        CodeGen<'ws,'w>
    abstract Add: Value<'ws> -> Value<'cnt> * Value<'idx>*Value<'idx> ->
        CodeGen<unit,'w>
    abstract Del: Value<'ws> -> Value<'cnt> * Value<'idx>*Value<'idx> ->
        CodeGen<unit,'w>
    abstract Pick: Value<'ws> -> Value<'cnt> ->
        CodeGen<'idx*'idx,'w>
    abstract HasMore: Value<'ws> -> Value<'cnt> ->
        CodeGen<bool,'w>
    abstract All: Value<'ws> -> Value<'cnt> ->
        CodeGen<seq<'idx*'idx>,'w>
```

Interface 5.31 (Expansion Strategy Module) The `ExpansionStrategy` module is parametrized by the type of indices `'idx`, the container `'cnt`, and the working set `'ws`. The open type `'exp` provides the access to the expansion strategy.

The only methods provided by the expansion strategy are the `Init` code generator and the `Expand` code generator. Given the basis container and the selection strategy, the code generated by `Expand` function expands the working set by the critical pairs that contain the newly added polynomial. The expansion strategy decides which critical pairs are superfluous and omits their inclusion in the working set. The strategy could take into consideration any signature-based expansion algorithms such as those defined by Sun and Wang [82].

```
type ExpansionStrategy<'idx,'cnt,'ws,'exp> =
    abstract Init: Value<'cnt> * Value<'ws> ->
           CodeGen<'exp,'w>
    abstract Expand: Value<'exp> -> Value<'cnt> * Value<'ws> * Value<'idx> ->
           CodeGen<unit,'w>
```

Interface 5.32 (S-Polynomial Module) The `SPoly` module is implemented as the open type `'sp` and is parametrized by the polynomial type `'poly` in the polynomial algebra, the choice of polynomial basis container `'cnt` and the indices of the elements `'idx`.

The methods provided by the `SPoly` module are an initialization generator routine `Init` and the σ code generator which, given the container and two polynomial indices within this container, computes the S-polynomial of the two polynomials. Note that this resultant S-polynomial is returned as a polynomial value `'poly` since it has no index in the container yet. It is the responsibility of the expansion algorithm to insert the normal form of this polynomial in the container in the future.

```
type SPoly<'poly,'idx,'cnt,'sp> =
    abstract Init: Value<'cnt> ->
           CodeGen<'sp,'w>
    abstract σ: Value<'sp> -> Value<'cnt> * Value<'idx>*Value<'idx> ->
           CodeGen<'poly,'w>
```

Interface 5.33 (Normal Remainder Module) The `NormalRemainder` module is a generic type `'nr` parametrized by the type of polynomials `'poly` and the basis container `'cnt` and provides a code generator `NR` which given a polynomial computes its normal form.

```
type NormalRemainder<'poly,'cnt,'nr> =
    abstract Init: Value<'cnt> ->
           CodeGen<'nr,'w>
```

```
    abstract NR: Value<'nr> -> Value<'cnt> * Value<'poly> ->
           CodeGen<'poly,'w>
```

Interface 5.34 (Reduction Strategy Module) The `ReductionStrategy` module of type `'rs` provides a code generator `Reduce` as a part of the post processing algorithm which, given an index of type `'idx` and its container of type `'cnt`, decides whether the given polynomial is reducible with respect to the other polynomials in the container. This module is only initialized and used when the control is computing a minimal or a reduced basis.

```
type ReductionStrategy<'idx,'cnt,'rs> =
    abstract Init: Value<'cnt> ->
           CodeGen<'rs,'w>
    abstract Reduce: Value<'rs> -> Value<'cnt> * Value<'idx> ->
           CodeGen<bool,'w>
```

Interface 5.35 (Canonical Form Module) The `CanonicalForm` module of type `'cf` provides a code generator `Canonicalize` as a part of the post processing algorithm that computes the canonical form of a given polynomial of type `'poly` with respect to the basis container of type `'cnt`. The control module only initializes and calls this module if computation of the reduced Gröbner bases is needed.

```
type CanonicalForm<'poly,'cnt,'cf> =
    abstract Init: Value<'cnt> ->
           CodeGen<'cf,'w>
    abstract Canonicalize: Value<'cf> -> Value<'cnt> * Value<'poly> ->
           CodeGen<'poly,'w>
```

We will see some implementations of these modules in §5.3.3 with examples of their usage.

5.3.2 Implementation of the Generation Algorithm

The code generator requires an implementation of the polynomial algebra and every one of the modules described in the last section to generate the instance of Buchberger's algorithm specialized for the described problem. The generator is able to produce many different instances of the algorithm by mixing and matching different implementations of each module. For example, the generation of the solver for Linear

Programming specialization shares all the modules in common with the full generic instance of Buchberger's algorithm with a different specialization of `Input` and `Output` modules. Similarly, multiple implementations of each algorithm can be generated by changing the choice of the polynomial algebra to reflect different fields of coefficients.

The code generator for the control modules requires two more parameters which we have not defined yet: The choice of which flavour of Gröbner bases is being generated, and generation of tracing and debugging information.

Definition 5.36 (Basis Kind) The type `BasisKind` is an input to the main controller of the code generator and describes which version of the algorithm is to be implemented. A value of `StandardBasis` only computes a Gröbner basis for the given input, whereas a value of `MinimalBasis` computed a minimal Gröbner basis by invoking the `ReductionStrategy` module, and a value of `ReducedBasis` computes the reduced Gröbner basis by invoking both the `ReductionStrategy` and `CanonicalForm` modules.

```
type BasisKind =
    | StandardBasis
    | MinimalBasis
    | ReducedBasis
```

Definition 5.37 (Trace Generation) We are introducing an additional module for debugging and tracing which generates print statements in the output program to report on the progress of the solver during the execution as well as printing information about selected critical pairs of polynomials and their resultants. In §5.3.3 we will see how this aspect module helps with both writing examples and as a teaching tool for Gröbner bases computation.

```
type Trace =
    abstract Level: int
    abstract Write: int -> Value<'a> ->
           CodeGen<unit,'w>
    abstract WriteLine: int -> Value<'a> ->
           CodeGen<unit,'w>
```

The one advantage of generating trace and debug statements at generation time instead of execution is that if the user opts not to have any debugging in the code, the generated algorithm is completely free of any conditional or printing statements that would have otherwise been deactivated.

Interface 5.38 (Gröbner Bases Solver Generator) The `GBSolver` function is the main code generator of this thesis that inputs all the modules and data defined in this thesis so far and generates a specialized solver program for finding the Gröbner bases defined by the input parameters. The output from this program is an expression which embodies the solver program. We will rely on the predicates Gröbner, Minimal and Reduced which check if a given basis is a Gröbner basis, minimal, and reduced respectively, as defined in §2.2.5.

$$
\begin{aligned}
&{}_p|\text{GBSolver}|_f \ \models \\
&\quad p = (\text{DB}, \text{BK}, \text{PA}, \text{IP}, \text{PC}, \text{SS}, \text{ES}, \text{SP}, \text{NR}, \text{RS}, \text{CF}, \text{OP}) \ \wedge \\
&\quad \text{DB} \ |\models \text{Trace} \ \wedge \\
&\quad \text{BK} \in \text{BasisKind} \ \wedge \\
&\quad \text{PA} \ |\models \text{PolynomialAlgebra} \ C, \ M, \ T, \ \mathsf{Poly} \ \wedge \\
&\quad \text{IP} \ |\models \text{Input} \ \mathsf{Poly}, \ \mathsf{Inp} \ \wedge \\
&\quad \text{PC} \ |\models \text{Container} \ \mathsf{Poly}, \ \mathsf{Idx}, \ \mathsf{Cnt} \ \wedge \\
&\quad \text{SS} \ |\models \text{SelectionStrategy} \ \mathsf{Idx}, \ \mathsf{Cnt}, \ \mathsf{Sel} \ \wedge \\
&\quad \text{ES} \ |\models \text{ExpansionStrategy} \ \mathsf{Idx}, \ \mathsf{Cnt}, \ \mathsf{Sel}, \ \mathsf{Exp} \ \wedge \\
&\quad \text{SP} \ |\models \text{SPoly} \ \mathsf{Poly}, \ \mathsf{Idx}, \ \mathsf{Cnt}, \ \mathsf{Sp} \ \wedge \\
&\quad \text{NR} \ |\models \text{NormalRemainder} \ \mathsf{Poly}, \ \mathsf{Cnt}, \ \mathsf{Nr} \ \wedge \\
&\quad \text{RS} \ |\models \text{ReductionStrategy} \ \mathsf{Idx}, \ \mathsf{Cnt}, \mathsf{Rs} \ \wedge \\
&\quad \text{CF} \ |\models \text{CanonicalForm} \ \mathsf{Poly}, \ \mathsf{Cnt}, \ \mathsf{Cf} \ \wedge \\
&\quad \text{OP} \ |\models \text{Output} \ \mathsf{Poly}, \ \mathsf{Out} \ \twoheadrightarrow \\
&\quad \bar{f} \in \ulcorner \mathsf{Inp} \ \rightarrow \ \mathsf{Out} \urcorner \ \wedge \\
&\quad \forall_{\sigma \in \Sigma \ \ i} |[\![f]\!]\sigma|_o \ \models \\
&\qquad i \subset C[\vec{x}] \ \twoheadrightarrow \\
&\qquad \bar{o} \subset C[\vec{x}] \ \wedge \\
&\qquad \langle i \rangle = \langle \bar{o} \rangle \ \wedge \\
&\qquad \text{Gröbner}(\bar{\text{o}}) \ \wedge \\
&\qquad \text{BK} \in \{\text{MinimalBasis}, \text{ReducedBasis}\} \ \Rightarrow \ \text{Minimal}(\bar{o}) \ \wedge \\
&\qquad \text{BK} = \text{ReducedBasis} \ \Rightarrow \ \text{Reduced}(\bar{o})
\end{aligned}
$$

Implementation 5.39 (Gröbner Bases Solver Generator)

```
let GBSolver(DB:Trace,
             BK:BasisKind,
             PA:PolynomialAlgebra<_,_,_,_>,
             IP:Input<_,_>,
             PC:Container<_,_,_>,
             WS:WorkingSet<_,_,_>,
             ES:ExpansionStrategy<_,_,_,_>,
             SP:SPoly<_,_,_,_>,
             NR:NormalRemainder<_,_,_>,
```

```
            RS:ReductionStrategy<_,_,_>,
            CF:CanonicalForm<_,_,_>,
            OP:Output<_,_>) =

    // Generative implementation of Buchberger's algorithm
    let gen f = codegen {
        do! StateRecord("Basis␣Kind").Extend BK

        // Initialize main modules
        let! _ = DB.WriteLine 1 <| V"Begin␣initialization"
        let! ip = IP.Process f
        let! pc = PC.Init ip
        let! ws = WS.Init pc
        let! es = ES.Init (pc,ws)
        let! sp = SP.Init pc
        let! nr = NR.Init pc
        let! _ = DB.WriteLine 1 <| V"Initialization␣complete"

        // Loop through all pairs of polynomials
        while WS.HasMore ws pc do
            // Pick a pair of polynomials
            use! p = WS.Pick ws pc
            let! _ = DB.Write 2 <| V"Picked␣pair:␣"
            let! _ = DB.WriteLine 2 p

            // Calculate the S-Polynomial
            use! σ = SP.σ sp (pc, Fst p, Snd p)
            let! _ = DB.Write 3 <| V"␣-␣Residual␣=␣"
            let! _ = DB.WriteLine 3 σ

            // Calculate the normal remainder of the s-poly
            use! nr_σ = NR.NR nr (pc,)
            let! _ = DB.Write 3 <| V"␣-␣Normalized␣=␣"
            let! _ = DB.WriteLine 3 nr_σ

            // If the new polynomial did not reduce to 0
            yield! IfU (Not (PA.isZero nr_σ)) <| codegen {
                // Add new polynomial
                use! j = PC.Add pc nr_σ
                let! _ = DB.Write 2 <| V"␣*␣Adding␣new␣polynomial␣#"
                let! _ = DB.WriteLine 2 j
```

```
            // Expand the set of polynomial pair according to the newly added element
            yield! ES.Expand es (pc,ws,j)
        }

    let! _ = DB.WriteLine 1 <| V"Begin␣post-processing"
    let! res =
        if BK = StandardBasis then PC.All pc
        else codegen {
            // Initialize modules for computing reduced/minimal Groebner bases
            let! rs = RS.Init pc
            let! cf = CF.Init pc
            let! idx = PC.Indexes pc
            let! _ = DB.WriteLine 1 <| V"Reductions␣initiated"

            // Reduce unnecessary polynomials (compute reduced GB)
            let! idx' = Iterate Seq.Filter idx <| fun i -> codegen {
                use! b = RS.Reduce rs (pc,i)
                let! _ = DB.Write 2 <| V"Polynomial␣#"
                let! _ = DB.Write 2 i
                let! _ = DB.WriteLine 2 <| Control.If b (V"␣is␣redundant") (V"␣
                    does␣not␣reduce")
                yield Not b
            }

            yield! Iterate Seq.Map idx' <| fun i -> codegen {
                let! c_i = PC.Get pc i
                yield!
                    if BK = MinimalBasis then Return c_i
                    else codegen {
                        // Canonicalize polynomials (compute minimal GB)
                        let! p = CF.Canonicalize cf (pc,c_i)
                        let! _ = DB.Write 3 <| V"␣-␣Canonicalized␣=␣"
                        let! _ = DB.WriteLine 3 p
                        yield p
                    }
            }
        }
    // The basis output
    let! _ = DB.WriteLine 1 <| V"Returning␣result"
    return! OP.Process res
}
```

```
gen |> Fun |> Generate
```

The type of this function is:

```
val GBSolver :
  Trace * BasisKind * PolynomialAlgebra<'a,'b,'c,'d> * Input<'d,'e> *
  Container<'d,'f,'g> * WorkingSet<'h,'g,'i> * ExpansionStrategy<'f,'g,'i,'j> *
  SPoly<'d,'h,'g,'k> * NormalRemainder<'d,'g,'l> *
  ReductionStrategy<'f,'g,'m> * CanonicalForm<'d,'g,'n> * Output<'d,'o> ->
  Value<('e -> 'o)>
```

which inputs the twelve choices for specializing the generator and outputs a specialized implementation of Buchberger's algorithm as a function of type `'e -> 'o`.

5.3.3 Sample Implementations

We start the first example usage of this code generator to construct a program that corresponds to Algorithm 2.56. Before doing so, we provide some implementations of the modules that are of general use, but also directly apply to this instance of the problem.

Implementation 5.40 (No Tracing) The simplest implementation of the `Trace` module produces no debug information.

```
module Debug =
    let NoTrace<'a> =
        { new Trace with
            member t.Level = 0
            member t.Write _ _ = Return Unit
            member t.WriteLine _ _ = Return Unit
        }
```

Implementation 5.41 (Direct I/O) The direct version of the `Input` and `Output` modules do not perform any data processing on the input and output, thereby making the types of the in/out data a sequence of polynomials to be directly processed by the main algorithm. Notice that these special modules "disappear" after the generation stage since they do not produce any code at all.

```
module Input =
    let InBasis<'c> =
        { new Input<'c,_> with
            member i.Process r = Return r
        }
```

```
module Output =
    let OutBasis<'c> =
        { new Output<'c,_> with
            member o.Process c = Return c
        }
```

Implementation 5.42 (Identity Post Process) The identity implementations of the `ReductionStrategy` and `CanonicalForm` modules perform no operations on the data. When the basis kind parameter is set to `StandardBasis`, the generator does not use these modules, but `null` is not an acceptable implementation for any module interfaces. Thus, we need to introduce an identity implementation of these two modules that perform no operation.

```
module ReductionStrategy =
    let NoReduction<'i,'c> =
        { new ReductionStrategy<'i,'c,_> with
            member s.Init _ = Return Unit
            member s.Reduce _ (_,_) = Return False
        }

module CanonicalForm =
    let NoOperation<'t,'c when 't: equality> =
        { new CanonicalForm<'t,'c,_> with
            member f.Init _ = Return Unit
            member f.Canonicalize _ (_,p) = Return p
        }
```

Implementation 5.43 (List Container) The `ListContainer` implementation of the `Container` module implements the polynomial container as a .NET `List` with integer indices.

```
module Container =
    let ListContainer<'p> =
        { new Container<'p,_,_> with
            member lc.Init s = Let <| Return (List.New s)
            member lc.Add l p = codegen {
                yield List.Add l p
                return Idx.sub (List.Count l) Idx.one
            }
            member lc.Get l i = Return <| List.Item l i
            member lc.All l = Return <| Seq.CastTo l
```

```
        member lc.Indexes l = codegen {
            let b = Idx.zero
            let e = Idx.sub (List.Count l) Idx.one
            return Seq.Make b e
        }
    }
```

Implementation 5.44 (Direct Critical Pair Selection) The simple implementation of the working set chooses critical pairs of polynomials in computation of Gröbner bases by exhaustively analysing every possible pair and compute the residuals. `DirectPick` implements this selection strategy by using a list of pairs of indices from the container. Other specializations of this module will be used later, such as the implementation of choosing the critical pair with the smallest lcm.

```
module WorkingSet =
    let DirectPick<'i,'c> =
        { new WorkingSet<'i,'c,_> with
            member s.Init _ = Let <| Return (List.Empty())
            member s.Add l (_,i,j) = codegen {
                return List.Add l (Pair i j)
            }
            member s.Del l (_,i,j) = codegen {
                return Control.Ignore(List.Remove l (Pair i j))
            }
            member s.HasMore l _ = codegen {
                return (List.Count l) ^> Idx.zero
            }
            member s.Pick l _ = codegen {
                use h = List.Item l Idx.zero
                yield List.RemoveAt l Idx.zero
                return h
            }
            member s.All l _ = Return <| Seq.CastTo l
    }
```

Implementation 5.45 (Direct Critical Pair Expansion) Similar to the direct selection strategy, the `DirectExpand` module implementation expands the working set of critical pairs by all the possible combinations of the new polynomials and the existing basis elements.

```
module ExpansionStrategy =
```

```
let DirectExpand(C:Container<_,_,_>,S:WorkingSet<_,_,_>) =
    { new ExpansionStrategy<_,_,_,_> with
        member e.Init(c,s) = Prepend <| codegen {
            let! l = C.Indexes c
            for i in Seq.AllPairs l do
                yield! S.Add s (c, Fst i, Snd i)
        }
        member e.Expand _ (c,s,j) = codegen {
            let! l = C.Indexes c
            for i in l do
                yield! IfU (i ^<> j) (S.Add s (c,i,j))
        }
    }
```

Implementation 5.46 (Generic S-Polynomials) The direct method of computing S-polynomials is to implement $\mathbf{S}_{\sigma,\rho_1,\rho_2}$ from Definition 2.51 using the methods provided by the polynomial algebra.

```
module SPoly =
    let GenericSPoly(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>) =
        { new SPoly<_,_,_,_> with
            member p.Init _ = Return Unit
            member p.σ _ (c,i,j) = codegen {
                let! c_i = C.Get c i
                let! c_j = C.Get c j
                let lm_i = A.LM c_i
                let lm_j = A.LM c_j
                let lc_i = A.LC c_i
                let lc_j = A.LC c_j
                let t1 = A.mul (A.fromTerm (A.CR.τ lc_i lc_j)
                                           (A.TM.MM.τ lm_i lm_j)) (A.RT c_i)
                let t2 = A.mul (A.fromTerm (A.CR.τ lc_j lc_i)
                                           (A.TM.MM.τ lm_j lm_i)) (A.RT c_j)
                return A.sub t1 t2
            }
        }
```

Implementation 5.47 (Polynomial Remainders) Similar to the generic S polynomial generator, the `PolyDivNR` implementation of `NormalRemainder` computes the normal remainders by using the division algorithm provided by the polynomial algebra

as explained in Definition 2.53. We may also specialize this module by implementing the normal remainders as a polynomial rewrite system as defined in §2.2.4.

```
module NormalRemainder =
    let Remainder (A:PolynomialAlgebra<_,_,_,_>) =
        let remainder rec_rem c p = Generate <| codegen {
            use p' = Seq.Fold (BinaryOp.Flatten A.rem) p c
            return Control.If (p' ^<> p) (Function.Apply2 rec_rem c p') p'
        }
        DefineOnceRec "remainder" (fun rem -> BinaryOp.Flatten <| remainder rem)

    let PolyDivNR(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>) =
        { new NormalRemainder<_,_,_> with
            member d.Init _ = Remainder A
            member d.NR n (c,p) = codegen {
                let! a = C.All c
                return Function.Apply2 n a p
            }
        }
```

Example 5.48 (Generated Buchberger's Algorithm) We can now generate Algorithm 2.56 using all the module implementations defined so far. This example generates an instance of Buchberger's algorithm that computes a standard basis (not minimal). The polynomials are over the field `QQ` as defined by Example 5.13 which is the implementation of the field of rationals $\mathbb{Q}$ using `BigRational` arithmetic. The polynomials are implemented as a list of generic terms over dense monomials, and the terms are ordered according to `Lex` term ordering. We will use a .NET list as the container of the basis elements and use no special optimization techniques in the computation of the Gröbner bases without generating any trace information:

```
let t = Debug.NoTrace
let bk = StandardBasis
let K = Field.QQ
let mg = MonomialMonoid.Dense
let tm = TermModule.Generic(K,mg)
let ord = TermOrder.Dense.Lex mg
let pa = PolynomialAlgebra.Generic(tm,ord)
let ip = Input.InBasis
let pc = Container.ListContainer
let ws = WorkingSet.DirectPick
let es = ExpansionStrategy.DirectExpand(pc,ws)
```

```
let sp = SPoly.GenericSPoly(pa,pc)
let nr = NormalRemainder.PolyDivNR(pa,pc)
let rs = ReductionStrategy.NoReduction
let cf = CanonicalForm.NoOperation
let op = Output.OutBasis
let gb = GBSolver(t,bk,pa,ip,pc,ws,es,sp,nr,rs,cf,op)
```

After execution, the following code is generated:

```
1  fun a_1 ->
2      let v_1 = new List<(BigRational * seq<int>) list> (a_1)
3      let v_2 = new List<int * int> ()
4      for i_1 in List.allPairs {0 .. v_1.Count - 1} do
5          v_2.Add (fst i_1, snd i_1)
6      let rec remainder i j =
7          let t_1 = Seq.fold (fun i j -> snd (longdiv i j)) j i
8          if t_1 <> j then remainder i t_1 else t_1
9      while v_2.Count > 0 do
10         let t_1 = v_2.[0]
11         let t_2 = v_2.RemoveAt 0; t_1
12         let t_3 = sub
13             (mul v_1.[fst t_2].Tail
14                  [(fst v_1.[snd t_2].Head),
15                   τ (snd v_1.[fst t_2].Head) (snd v_1.[snd t_2].Head)])
16             (mul v_1.[snd t_2].Tail
17                  [(fst v_1.[fst t_2].Head),
18                   τ (snd v_1.[ snd t_2].Head) (snd v_1.[fst t_2].Head)])
19         let t_4 = remainder v_1 t_3
20         if not t_4.IsEmpty then
21             let t_5 = v_1.Add t_4; v_1.Count - 1
22             for i_1 in 0 .. v_1.Count - 1 do
23                 if i_1 <> t_5 then v_2.Add (i_1, t_5)
24     v_1
```

This is a printout of the code generated directly by the code generator. The printout has been edited for presentation purposes, but the content has not changed. We made the following changes to the output of the program presented above:

- The beginning of the lines have been indented by spaces to improve readability.
- Some statements have been broken down in multiple lines to fit the width of the page. Lines $12 - 18$ are an example of a line for which we had to insert line breaks.

- Extraneous parentheses have been removed to improve readability. For example, in line 19 the original statement generated is:
  `let t_4 = (((remainder)(v_1))(t_3))`

- Lists are compiled and generated using the `::` list constructor. We replaced these instances with the equivalent inline syntax of `[...]` for presentation.

With the exception of the syntactic sugar listed above, the presented code is exactly as provided by the code generator.

GBSolver Result	**Buchberger's Algorithm**	**Description**
Line 1	Line 1	Parameter `a_1` is the input F to the program.
Line 2	Line 2	Variable `v_1` is the Gröbner basis G, initialized to the input F (`a_1`).
Lines 3-5	Line 3	Variable `v_2` is the working set B, initialized to all the pairs of elements from G (`v_1`).
Lines 6-8	—	Introducing function `remainder` that computes normal remainders NR_G.
Line 9	Line 4	Loops until the working set B (`v_2`) is empty.
Lines 10,11	Lines 5,6	Picks out the variable `t_2` as the critical pair $\langle i, j\rangle$ from the working set B (`v_2`).
Lines 12-18	Line 7	Computes the S-polynomial $\sigma = \mathbf{S}_{G_i,G_j}$ of the critical pair $\langle i, j\rangle$ stored in `t_2` as the variable `t_3`.
Line 19	Line 8	Computes the normal remainder (`remainder`) $s = \mathrm{NR}_G\sigma$ of `t_3` as the variable `t_4`.
Line 20	Line 9	Checks if the residual s (`t_4`) is non-zero.
Line 21	Line 10	Adds the new polynomial s (`t_4`) to the computed basis G (`v_1`).
Lines 22,23	Line 11	Adds all the new critical pairs that use s (`t_4`) to the working set B (`v_2`).
Line 24	Line 12	Returns the computed Gröbner basis G (`v_1`).

Figure 5.8: Comparison of Algorithm 2.56 and the result of `GBSolver` code generator

Figure 5.8 shows a line-by-line comparison of this generated code and the statement of Buchberger's Algorithm in §2.3.
□

To implement the improved Buchberger's algorithm outlined in Algorithm 2.61, we need to introduce different selection and expansion strategies which implement Buchberger's two criteria.

Implementation 5.49 (Buchberger's Criteria) A specialization of the expansion strategy which implements Buchberger's two criteria as shown in Lemmas 2.59 and 2.60 is provided as the implementation `ExpandBuchbergerTriples`. This implementation is specialized to: (1) only expand by critical pairs whose leading monomials are relatively prime (i.e. do not satisfy Buchberger's first criterion), and (2) do not expand the working set by any Buchberger triples that satisfy Buchberger's second criterion.

```
module ExpansionStrategy =
    ⋮
    let InTriple (A:PolynomialAlgebra<_,_,_,_>) =
        let in_triple dic ids k i = codegen {
            use t_i = Dictionary.Item dic i
            let! test = Iterate Seq.Forall ids <| fun j -> codegen {
                use t_j = Dictionary.Item dic j
                use lcm = A.TM.MM.lcm t_i t_j
                return (j ^= i)
                   ^|| ((j ^< i) ^&& (lcm ^<> t_i))
                   ^|| ((j ^> i) ^&& (lcm ^<> t_j))
            }
            return (i ^<> k) ^&& test
        }
        DefineOnce "in_triple" <| Generate(Fun4 in_triple)

    let ExpandBuchbergerTriples(
        A:PolynomialAlgebra<_,_,_,_>,
        C:Container<_,'idx,_>,
        S:SelectionStrategy<_,_,_>) =
        { new ExpansionStrategy<_,_,_,_> with
            member e.Init(c,s) = codegen {
                let! in_triple = InTriple A
                return! PrependV in_triple <| codegen {
                    let! l = C.Indexes c
                    for i in Seq.AllPairs l do
                        yield! S.Add s (c, Fst i, Snd i)
                }
            }
```

```
        member e.Expand es (c,s,k) = codegen {
            use! ids = C.Indexes c
            use dic = Dictionary.New()
            let! c_k = C.Get c k
            use p = A.LM c_k
            for i in ids do
                let! c_i = C.Get c i
                yield Dictionary.Add dic i (A.TM.MM. p (A.LM c_i))
            use a = Seq.Filter (Function.Apply3 es dic ids k) ids
            let! pairs = S.All s c
            let filter p = codegen {
                let! t1 = Iterate Seq.Exists a <| fun i ->
                                  Return(i ^<> (Fst p))
                let! t2 = Iterate Seq.Exists a <| fun i ->
                                  Return(i ^<> (Snd p))
                return t1 ^&& t2
            }
            use! d = Iterate Seq.Filter (Seq.ToList pairs) filter
            for i in d do
                yield! S.Del s (c, Fst i, Snd i)
            for i in a do
                yield! S.Add s (c,i,k)
        }
    }
```

Implementation 5.50 (Least LCM Selection) This specialization of the working set improves on the optimization of the algorithm by giving selection priority to critical pairs whose leading monomials have the smallest least common multiple. These critical pairs are more likely to lead to new basis elements as shown by Gebauer and Möller [39].

```
module WorkingSet =
    ⋮
    let MakeComparer f1 f2 =
        { new IComparer<_> with
            member c.Compare(x,y) = match f1 x y with 0 -> f2 x y | n -> n }
    let LeastLCMPick(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>) =
        { new WorkingSet<_,_,_> with
            member s.Init _ = Let <| codegen {
                let f1 x y = A.cmp (Fst x) (Fst y)
                let f2 x y = Compare (Snd x) (Snd y)
```

```
            let cmp = BinaryOp.Lift MakeComparer <@MakeComparer@>
            return SortedSet.New <|
                   cmp (BinaryOp.Flatten f1) (BinaryOp.Flatten f2)
        }
        member s.Add l (c,i,j) = codegen {
            let! c_i = C.Get c i
            let! c_j = C.Get c j
            let lcm = A.TM.MM.lcm (A.LM c_i) (A.LM c_j)
            let t = Pair lcm (Pair i j)
            return Control.Ignore <| SortedSet.Add l t
        }
        member s.Del l (c,i,j) = codegen {
            let! c_i = C.Get c i
            let! c_j = C.Get c j
            let lcm = A.TM.MM.lcm (A.LM c_i) (A.LM c_j)
            let t = Pair lcm (Pair i j)
            return Control.Ignore <| SortedSet.Remove l t
        }
        member s.HasMore l _ = codegen
            return (SortedSet.Count l) ^> Idx.zero
        }
        member s.Pick l _ = codegen {
            use h = SortedSet.Min l
            yield Control.Ignore <| SortedSet.Remove l h
            return Snd h
        }
        member s.All l _ = Return <| Seq.Map (UnaryOp.Flatten Snd) l
    }
```

Additionally, to perform the post-processing operations for the generation of minimal and reduced Gröbner bases, we have the following two implementations of the reduction strategy and canonical forms.

Implementation 5.51 (Minimal Gröbner Basis) This specialization of the reduction strategy module computes the minimal Gröbner basis by divisibility of the leading terms (as provided by the polynomial algebra) between the computed basis elements to reduce the superfluous polynomials.

```
module ReductionStrategy =
    ⋮
    let EliminateDivisors(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>) =
```

```
{ new ReductionStrategy<_,_,_> with
    member s.Init c = Let <| codegen {
        let! idx = C.Indexes c
        return List.New idx
    }
    member s.Reduce rs (c,i) = codegen {
        let! c_i = C.Get c i
        use lm_i = A.LM c_i
        let! loop = Iterate Seq.Exists rs (fun j ->
            If (j ^= i)
                (Return False) <|
                codegen {
                    let! c_j = C.Get c j
                    use lm_j = A.LM c_j
                    let lcm = A.TM.MM.lcm lm_i lm_j
                    return lcm ^= lm_i
                })
        yield Control.If loop (List.Remove rs i) False
    }
}
```

Implementation 5.52 (Reduced Gröbner Basis) As the last stage of the post processing operations, to compute the reduced Gröbner basis the `ReducedBasis` specialization of the canonical form module first creates a monic polynomial from every computed basis element by scaling the entire polynomial by the inverse of the leading coefficient as provided by the polynomial algebra. To compute this inverse, we require the coefficients of the polynomials to form a field structure. It is possible that another specialization of this module performs fraction-free scaling of the elements using a quotient ring instead of a field.

Secondly, this module generates a canonicalization function in the output code that performs the reduction algorithm on every term in the basis elements using polynomial division. It is possible for another specialization of this module to provide the same canonicalization using a polynomial rewrite system instead.

```
module CanonicalForm =
    ⋮
    let Canonicalize (A:PolynomialAlgebra<_,_,_,_>) =
        let canonicalize rec_can remainder c p = Generate <| codegen {
            return! If (A.isZero p) (Return p) <| codegen {
                use rt = A.RT p
```

```
            let nrt = Function.Apply2 remainder c rt
            use rt' = Function.Apply2 rec_can c nrt
            return A.add (A.sub p rt) rt'
        }
    }
    codegen {
        let! remainder = NormalRemainder.Remainder A
        yield! DefineOnceRec "canonicalize" (fun rec_can ->
                BinaryOp.Flatten <| canonicalize rec_can remainder)
    }

let ReducedBasis(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>) =
    { new CanonicalForm<_,_,_> with
        member f.Init _ = Canonicalize A
        member f.Canonicalize r (c,p) = codegen {
            let! a = C.All c
            use cf = Function.Apply2 r a p
            return A.scalar (Field.Inverse A.TM.CR (A.LC cf)) cf
        }
    }
```

Example 5.53 (Trace Generation) This example demonstrates the usefulness of the trace generation as both a debugging tool and an illustration and teaching method. We show in this example how the samples in §2.3.5 were actually entirely generated using this program!

We will set up the parameters to the generator with the same values as Example 5.48, with the difference of generating the reduced Gröbner basis by setting the basis kind to `ReducedBasis` and providing the modules `ReductionStrategy.EliminateDivisors` and `CanonicalForm.ReducedBasis` for post processing operations. Additionally, we define and use the following `Trace` module:

```
module Debug =
    ⋮
let ConsoleTrace cap =
    { new Trace with
        member t.Level = cap
        member t.Write lvl s =
            if lvl > cap then Return Unit
            else print <@ System.Console.Write @> s |> Return |> Prepend
        member t.WriteLine lvl s =
```

```
if lvl > cap then Return Unit
else print <@ System.Console.WriteLine @> s |> Return |> Prepend
```

Let the function `gb` be the result of this code generator with trace level 5, and let function `makepoly` be a function of type `(BigRational * #seq<int>)list -> term< BigRational,seq<int>> list` that produces the sample polynomials:

```
let f1 = makepoly[1N,[2;0;0]; 1N,[0;1;0]; 1N,[0;0;1]; -1N,[0;0;0]]
let f2 = makepoly[1N,[1;0;0]; 1N,[0;2;0]; 1N,[0;0;1]; -1N,[0;0;0]]
let f3 = makepoly[1N,[1;0;0]; 1N,[0;1;0]; 1N,[0;0;1]; -1N,[0;0;0]]
```

These values correspond to the polynomials

$$f_1 = x^2 + y + z - 1, \qquad f_2 = x + y^2 + z - 1, \qquad f_3 = x + y + z - 1$$

from §2.3.5. The result of executing `gb [f1;f2;f3]` produces the following transcript:

```
> gb [f1;f2;f3];;
Begin initialization
Initialization complete
Picked pair: (0, 1)
 - Residual = [(-1N,[1;2;0]); (-1N,[1;0;1]); (1N,[1;0;0]);
               (1N,[0;1;0]); (1N,[0;0;1]); (-1N,[0;0;0])]
 - Normalized = [(1N,[0;4;0]); (2N,[0;2;1]); (-2N,[0;2;0]);
                 (1N,[0;1;0]); (1N,[0;0;2]); (-1N,[0;0;1])]
 * Adding new polynomial #3
Picked pair: (0, 2)
 - Residual = ...
 - Normalized = [(1N,[0;3;0]); (1N,[0;2;1]); (-1N,[0;2;0]);
                 (1N,[0;1;1]); (1N,[0;0;2]); (-1N,[0;0;1])]
 * Adding new polynomial #4
Picked pair: (1, 2)
 - Residual = ...
 - Normalized = [(1N,[0;2;0]); (-1N,[0;1;0])]
 * Adding new polynomial #5
Picked pair: (0, 3)
 - Residual = ...
 - Normalized = []
Picked pair: (1, 3)
 - Residual = ...
 - Normalized = []
Picked pair: (2, 3)
 - Residual = ...
 - Normalized = [(-2N,[0;1;1]); (-1N,[0;0;2]); (1N,[0;0;1])]
```

```
 * Adding new polynomial #6
Picked pair: (0, 4)
 - Residual = ...
 - Normalized = [(-1/2N,[0;0;3]); (1/2N,[0;0;1])]
 * Adding new polynomial #7
Picked pair: (1, 4)
 - Residual = ...
 - Normalized = []
⋮
Begin post-processing
Reductions initiated
Returning result
Polynomial #0 is redundant
Polynomial #1 is redundant
Polynomial #2 does not reduce
 - Canonicalized = [(1N,[1;0;0]); (1N,[0;1;0]); (1N,[0;0;1]); (-1N,[0;0;0])]
Polynomial #3 is redundant
Polynomial #4 is redundant
Polynomial #5 does not reduce
 - Canonicalized = [(1N,[0;2;0]); (-1N,[0;1;0])]
Polynomial #6 does not reduce
 - Canonicalized = [(1N,[0;1;1]); (1/2N,[0;0;2]); (-1/2N,[0;0;1])]
Polynomial #7 does not reduce
 - Canonicalized = [(1N,[0;0;3]); (-1N,[0;0;1])]
```

This produces the three examples in §2.3.5 step by step. The monic polynomials in that section were produced by a different specialization of the canonical form module, `CanonicalForm.NormalizePoly`, which only puts the basis elements in monic form and does not perform any back-propagation.
□

Finally, it would be worthwhile to demonstrate how many variations of Buchberger's algorithm we are able to generate using only the module specializations provided in this chapter. There are many more possible specializations of these modules possible and we are only demonstrating the variations that produce a *full* version of Buchberger's algorithm using *generic* multi-variate polynomials. The special forms of polynomials and generation of sub-algorithms will be the subject of the next chapter.

Figure 5.9 shows all the combinations of the modules that we used to generate 221,184 different instances of Buchberger's algorithm.

Module	Count	Description
`Field`	6	One implementation of integers modulo a prime and two implementations of rationals, plus three implementations of complex numbers.
`MonomialMonoid`	2	Dense and Sparse representations
`TermModule`	1	Generic terms
`TermOrder`	4	`Lex`, `RevLex`, `DegLex`, and `DegRevLex` orderings
`PolynomialAlgebra`	1	Generic polynomials
`Input`	4	Identity input, constant input basis, reading from files (or keyboard), converting a matrix to set of linear polynomials
`Output`	3	Identity output, writing to files (or screen), converting linear polynomials to a coefficient matrix
`Container`	3	Linear storage with indices as polynomials, `List` storage, out-of-memory storage (file or database) with row ids as index
`SelectionStrategy`	2	`DirectPick` and `LeastLCMPick`
`ExpansionStrategy`	4	Direct expanding, or using Buchberger criteria 1, 2, or both
`Spoly`	1	`GenericSPoly`
`NormalRemainder`	2	Using polynomial division with remainder, or using a polynomial rewrite system
`ReductionStrategy`	2	No reduction (standard basis), or eliminating the redundant polynomials
`CanonicalForm`	4	No operation (minimal or standard basis), producing only monic polynomials, or full reduction of basis using either polynomial division or a rewrite system

Figure 5.9: Specializations Provided For Each Module

Chapter 6

Specializations of Gröbner Bases Computation Algorithms

There are many specialized algorithms that can be generated by the code generator from §5 as depicted by Figure 5.9, but we mentioned earlier that any sub-algorithm of Buchberger's algorithm may also be specialized and generated using this code generator. In this chapter we analyse two of the most common specializations of the algorithm: Gaussian Elimination and Euclidean Algorithm.

6.1 Gaussian Elimination

We originally introduced the Gaussian Elimination specialization of Buchberger's algorithm as finding a Gröbner basis for a set of linear polynomials. We first need to explore what properties this choice of linear polynomials entails, and then specialize the related modules in our code generator to reflect these definitions. Finally, we compare the generated algorithm to the actual statement of Gaussian Elimination.

6.1.1 Properties of Linear Polynomials

We first analyse the properties of choosing a linear system of polynomials represented as the polynomial algebra behind the computation of Gröbner bases. In this section we will define a representation method for linear polynomials analogous to matrices used in Gaussian Elimination and draw parallels between polynomials computations and row operations performed in matrix computations. We also briefly demonstrate

how Buchberger's algorithm performs in cubic time complexity when given this choice of polynomials.

Assume we have a system of m linear polynomials in n variables

$$p_j = \sum_{i=0}^{n-1} a_{j,i}x_i + a_{j,n} \text{ , for } j = 0, \ldots, m-1$$

with the $\sigma = \text{lex}$ term ordering, i.e. $i < j \Leftrightarrow x_i > x_j$, i.e., $x_0 > x_1 > \cdots > x_{n-1}$.

Definition 6.1 For each j, $0 \leq j < n$, the polynomial p_j is represented as the row vector $\vec{a_j} = [a_{j,0}; \cdots ; a_{j,n}]$. Let $\vec{x}$ be the column vector

$$\vec{x} = \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \\ 1 \end{bmatrix},$$

then

$$p_j = \vec{a_j} \cdot \vec{x} = \sum_{i=0}^{n-1} a_{j,i}x_i + a_{j,n}.$$

This immediately leads to the $m \times n + 1$ matrix $A = (a_{i,j})$ and the system of equations is represented by $A \times \vec{x}$. In this case, performing a computation with two polynomials p_i and p_j can be re-interpreted as performing a row operation on the rows $\vec{a_i}$ and $\vec{a_j}$, respectively. The solution of the system of equations is isomorphic to computing the Gaussian Elimination of the matrix A. The algebraic variety formed from the ideal $I = \langle p_0, \ldots, p_{m-1} \rangle$ is the set of points in $\mathbb{R}^n$ where $p_i(\vec{x}) = 0$ for all i, which is the set of solutions to the system of equations mentioned earlier. Therefore we can use Buchberger's algorithm together with elimination theorems to achieve the same result as solving the system of equations.

The following results directly compare polynomial computations to row operations:

Definition 6.2 Let P_t be the subset of $\{p_0, \ldots, p_{m-1}\}$ whose leading monomial is x_t.

We will use this definition for groups of polynomials throughout the computation of a Gröbner basis for the linear system given above. Note that given a polynomial group P_t, all the polynomials in this group share the same leading monomial x_t. i.e., the first non-zero entry in the vector representation of every polynomial is on column t.

Lemma 6.3 At any stage in computation of the Gröbner basis, given the pair $\langle f, g\rangle$ of polynomials to consider;

$$f \in P_t \wedge g \in P_s \wedge p \neq s \Rightarrow \mathrm{NR}_{\sigma,G}\mathbf{S}_{f,g} = 0$$

Proof $\mathrm{LM}_\sigma f = x_t$ since $f \in P_t$; similarly, $\mathrm{LM}_\sigma g = x_s$ since $g \in P_s$. Therefore, $\mathrm{lcm}(\mathrm{LM}_\sigma f, \mathrm{LM}_\sigma g) = x_t x_s$ since $x_t \neq x_s$.
By Lemma 2.59, $\mathbf{S}_{f,g} \overset{G}{\twoheadrightarrow} 0$. Since the algorithm only considers polynomials that are already placed in G, then we know that $\{f, g\} \subseteq G$. Therefore, $\mathrm{NR}_G\mathbf{S}_{f,g} = 0$.
□

Lemma 6.4 Given a pair of polynomials $f, g \in P_t$, assume $\mathbf{S}_{f,g} \in P_s$, then either $s > t$ or $\mathbf{S}_{f,g} = 0$. Meaning the S-polynomial of two polynomials in the same group is a member of a strictly lower ordered group.

Proof Let $f = \sum_{i=t}^{n-1} b_i x_i + b_n$ and $g = \sum_{i=t}^{n-1} c_i x_i + c_n$ for constant coefficient vectors $\vec{b}, \vec{c}$.
We will rewrite the two formulas to make the leading terms stand out:

$$f = b_t x_t + \sum_{i=t+1}^{n-1} b_i x_i + b_n, \qquad g = c_t x_t + \sum_{i=t+1}^{n-1} c_i x_i + c_n$$

$\mathrm{LM}_\sigma f = x_t$, $\mathrm{LM}_\sigma g = x_t$, then $\mathrm{lcm}(\mathrm{LM}_\sigma f, \mathrm{LM}_\sigma g) = x_t$

$$\mathbf{S}_{f,g} = \frac{\mathrm{lcm}(\mathrm{LM}_\sigma f, \mathrm{LM}_\sigma g)}{\mathrm{LT}_\sigma f} f - \frac{\mathrm{lcm}(\mathrm{LM}_\sigma f, \mathrm{LM}_\sigma g)}{\mathrm{LT}_\sigma g} g$$

$$= \frac{x_t}{b_t x_t}(b_t x_t + \sum_{i=t+1}^{n-1} b_i x_i + b_n) - \frac{x_t}{c_t x_t}(c_t x_t + \sum_{i=t+1}^{n-1} c_i x_i + c_n) = \sum_{i=t+1}^{n-1} (\frac{b_i}{b_t} - \frac{c_i}{c_t}) x_i + \frac{b_n}{b_t} - \frac{c_n}{c_t}$$

The leading monomial of the resulting S-polynomial is *at most* x_{t+1}. In the case of $\frac{b_i}{b_t} - \frac{c_i}{c_t} = 0$ then we have the S-polynomial as a member of an even lower ordered group P_{t+2}. If the chain of nil coefficients holds for all the variables $x_{t+1} \ldots x_n$, then $\mathbf{S}_{f,g} = 0$.
□

Lemma 6.3 shows that given two polynomials p_j, p_k that do not share the same leading monomial

$$\begin{array}{rcl} p_j & = & [0; \cdots; a; \cdots] \\ p_k & & [0; \cdots; 0; b; \cdots] \end{array}$$

the result of row operation does not change any rows.

Similarly, Lemma 6.4 shows that given two polynomials p_j, p_k which have the same leading monomial x_t,

$$\begin{array}{rcl} p_j & = & [0; \cdots ; a; \cdots] \\ p_k & & [0; \cdots ; b; \cdots] \end{array}$$

then the row operation of Gaussian elimination produces a new row which has a zero entry on column t:

$$\begin{array}{rcl} p_j & = & [0; \cdots ; a; \cdots] \\ p'_k & & [0; \cdots ; 0; \cdots] \end{array}$$

These results also confirm that the minimal Gröbner basis of P has the row-echelon form. Note that the reduction Algorithm 2.58 is exactly the back-propagation operation performed during a full Gaussian Elimination.

Now let us study what happens in the stage in Buchberger's algorithm where we are adding new pairs of polynomials to the set B. By Lemma 6.3 we know that we are only considering polynomials that belong to the same group P_t, otherwise Buchberger's Criterion 1 is false and no computation is performed.

We also know that due to Lemma 6.4 the S-polynomial of two elements taken from the same group P_t belongs to a lower ranked group P_s, $s > t$, and that the pairs formed from this new polynomial and any members of G that does not belong to P_s would fail the first Buchberger Criterion and need not be considered.

The following lemma helps us determine a bound on the number of iterations in Buchberger's algorithm:

Lemma 6.5 Let f, g be linear polynomials, then either $f \overset{g}{\twoheadrightarrow} \mathbf{S}_{f,g}$ if f and g share the same leading monomial, or $f \overset{g}{\twoheadrightarrow} f$ otherwise.

Proof Let $b_t \cdot x_t$ be the leading term of f and $c_s \cdot x_s$ be the leading term of g. If $t \neq s$ then the leading terms of f is not divisible by g and thus $\mathrm{NR}_g f = f$. The second statement of the lemma trivially holds. Assume that $s = t$, then $\mathbf{S}_{f,g} = f - \frac{b_t}{c_s} g$. By definition of normal remainder, we also have that $f = \frac{b_t}{c_s} g + \mathrm{NR}_g f$, then $\mathrm{NR}_g f = f - \frac{b_t}{c_s} g = \mathbf{S}_{f,g}$. $\therefore f \overset{g}{\twoheadrightarrow} \mathbf{S}_{f,g}$.
□

Corollary 6.6 There are at most n new polynomials computed during the execution of Buchberger's algorithm

Proof By Lemma 6.5, we know that given a polynomial f and a basis G, $\mathrm{NR}_G f$ can never share a leading monomial with any other polynomials in G. Since there is at most n possible leading monomials in the algebra of linear polynomials, there are at most n new polynomials computed by the main loop.
□

To demonstrate the use of these results, let us consider the following example of linear equations:

Example 6.7 Consider the following augmented matrix for Gaussian Elimination and its equivalent system of linear polynomials:

$$A_0 = \left[\begin{array}{cccc|c} 2 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & 0 & 0 \\ 0 & 3 & -2 & 1 & -4 \\ 1 & 3 & 0 & -2 & -3 \end{array}\right], \qquad \begin{cases} p_1 : 2x - y - z - w - 1 \\ p_2 : x - y - z \\ p_3 : 3y - 2z + w - 4 \\ p_4 : x + 3y - 2w - 3 \end{cases}$$

With term ordering $x > y > z > w$ and the field of coefficients $\mathbb{Q}$. Let $G = \{p_1, p_2, p_3, p_4\}$ at the beginning of the algorithm. Since Lemma 6.5 shows that the S-polynomial computation essentially performs the same computation as normal remainder, we will refrain from performing any NR reductions to demonstrate the row operations.

At first step, we compute the residuals (S-polynomials) of the first group $P_1 = \{p_1; p_3; p_4\}$:

$$A_1 = \left[\begin{array}{cccc|c} 2 & -1 & -1 & -1 & -1 \\ 0 & 1 & 1 & -1 & -1 \\ 0 & 3 & -2 & 1 & -4 \\ 0 & -7 & -1 & 3 & 5 \end{array}\right], \qquad \begin{cases} p_5 : y + z - w - 1 \\ p_6 : -7y - z + 3w + 5 \end{cases}$$

Similarly, performing the S-polynomial computations on the next group $P_2 = \{p_5; p_3; p_6\}$ lead to:

$$A_2 = \left[\begin{array}{cccc|c} 2 & -1 & -1 & -1 & -1 \\ 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & -5 & 4 & -1 \\ 0 & 0 & -6 & 4 & 2 \end{array}\right], \qquad \begin{cases} p_7 : -5z + 4w - 1 \\ p_8 : -6z + 4w + 2 \end{cases}$$

And the operation on $P_3 = \{p_7; p_8\}$ computes:

$$A_3 = \left[\begin{array}{cccc|c} 2 & -1 & -1 & -1 & -1 \\ 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & -5 & 4 & -1 \\ 0 & 0 & 0 & -4 & 16 \end{array}\right] \quad , \quad \left\{ \begin{array}{l} \\ \\ \\ p_9 : -4w + 16 \end{array} \right.$$

At this point there are no more compatible critical pairs to consider according to Lemma 6.3, therefore there are no more pairs in the working set and the matrix is in row-echelon form. We may continue to minimalize the computed basis, which produces monic polynomials:

$$A'_3 = \left[\begin{array}{cccc|c} 1 & -1/2 & -1/2 & -1/2 & -1/2 \\ 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & -4/5 & 1/5 \\ 0 & 0 & 0 & 1 & -4 \end{array}\right] \quad , \quad \left\{ \begin{array}{l} p'_1 : x - 1/2y - 1/2z - 1/2w - 1/2 \\ p'_5 : y + z - w - 1 \\ p'_7 : z - 4/5w + 1/5 \\ p'_9 : w - 4 \end{array} \right.$$

And finally, computing the reduced Gröbner basis completes the final back propagation step of the Gaussian Elimination algorithm:

$$A''_3 = \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & -5 \\ 0 & 1 & 0 & 0 & -2 \\ 0 & 0 & 1 & 0 & -3 \\ 0 & 0 & 0 & 1 & -4 \end{array}\right] \quad , \quad \left\{ \begin{array}{l} p''_1 : x - 5 \\ p''_5 : y - 2 \\ p''_7 : z - 3 \\ p''_9 : w - 4 \end{array} \right.$$

□

By removing the computation of normal remainders in the previous example, we have simplified the algorithm but also, lost the desirable result of Corollary 6.6. It is now possible to compute a new reduction of all m many polynomials for each polynomial group $P_1, \cdots, P_n$. This gives us a new bound of at most $m \cdot n$ many iterations of the main loop. Considering that the S-polynomial computation is an $\mathcal{O}(n)$ operation, we directly arrive at the following result:

Theorem 6.8 Buchberger's algorithm on a set of m linear polynomials of n variables has the time complexity $\mathcal{O}(m \cdot n^2)$.

We will later use all the properties learned in this section in §6.1.3 to specialize Buchberger's algorithm and generate a program that performs Gaussian Elimination.

6.1.2 Linear Polynomial Algebra

A *linear* polynomial is a multivariate polynomials of maximum degree one, i.e., every monomial may contain at most one variable whose exponent is non-zero, and that exponent may only be 1. We will define a polynomial algebra as per Interface 5.23 for linear polynomials to be used in the Gaussian Elimination specialization of our software — more specifically, this algebra can be used for any other specializations related to linear algebra.

Earlier in §5.1 we mentioned the importance of partial operators in algebraic objects. This becomes especially useful in case of linear polynomials since many operations on linear polynomials do not necessarily lead to another linear polynomial. For example, the product of two linear polynomials $f_1 = x + 1$ and $f_2 = y + 1$ is not a linear polynomial[1] and thus multiplication is partial for these values, but product of $f_1 = x + 1$ and $f_3 = 2$ is $f_1 \cdot f_3 = 2x + 2$.

We will leave most of the operations in the algebra of linear polynomials undefined for simplicity and only define the few operations that are required for this specialization. We will implement the linear polynomials of n variables as a `list` of $n + 1$ coefficients as defined by Definition 6.1. The ring of polynomials may be defined as:

```
let mkConst n z v = List.Append (List.Replicate (V n) z) (List.Singleton v)

let PolyRing n (c:#Ring<_>) =
    QuotientRing.Gen(
        mkConst n c.zero c.zero,
        mkConst n c.zero c.one,
        List.Map (UnaryOp.Flatten c.neg),
        List.Map2 (BinaryOp.Flatten c.add),
        (fun x y -> failwith "Undefined␣multiplication␣operation"),
        List.Map2 (BinaryOp.Flatten c.sub) |> Some,
        None, None, None, None, None, None)
```

Notice that all of the functions for division, remainder, gcd, etc. are unimplemented.

We will implement monomials as integer indexes such that for $i = 0 \cdots n - 1$, the monomial i represents x_i and the monomial n represents the zero monomial.

```
let Monomials n =
    MonomialMonoid.Gen(
```

[1] Using traditional definition of polynomial multiplication, this product is $f_1 \cdot f_2 = xy + x + y + 1$, which is not a linear polynomial.

```
    V n,
    Min,
    (fun l -> If (l ^= (V n)) Idx.zero Idx.one),
    (fun l -> Seq.Map (UnaryOp.Flatten(fun i -> If (l ^= i) Idx.one Idx.zero))
                      (Seq.Make Idx.zero (V(n-1)))),
    (fun l -> If (l ^= (V n)) (Seq.Empty())
                              (Seq.Singleton(Pair l Idx.one))),
    (fun l -> Seq.FindIndex (UnaryOp.Flatten(Bool.lt Idx.zero))
                            (Seq.Append l (Seq.Singleton Idx.one))),
    (fun l -> If (Seq.IsEmpty l) (V n) (Fst <| Seq.Head l)),
    (fun x y -> If (x ^= y) (Option.Some (V n)) (V None)),
    (fun x y -> If (x ^= y) x (V n)),
    (fun x y -> If (x ^= y) x (Idx.neg Idx.one)),
    (fun x y -> If (x ^= y) (V n) (Idx.neg Idx.one)))
```

Finally, the algebra of linear polynomials uses the partial polynomial ring defined earlier and provides functions (such as leading monomial or term) that link this representation of the polynomials to the monomial representation defined above.

```
let LinearPolynomial n c =
    let tm = TermModule.Generic(c, Monomials n)
    let pr = PolyRing n c
    let notzero = UnaryOp.Flatten (Bool.neq c.zero)
    PolynomialAlgebra.Gen(
        c, tm, pr,
        mkConst n c.zero,
        Some(fun s -> List.Map (UnaryOp.Flatten (c.mul s))),
        Compare,
        (fun p -> Seq.Mapi (BinaryOp.Flatten(fun i n -> tm.make n i)) p),
        None,
        List.Forall (UnaryOp.Flatten(Bool.eq c.zero)),
        (fun p -> Generate <| codegen {
            use i = List.FindIndex notzero p
            return tm.make (List.Nth p i) i
        }),
        Some(List.FindIndex notzero),
        Some(List.Find notzero),
        (fun p -> List.SetNth p (List.FindIndex notzero p) c.zero),
        Seq.Fold (BinaryOp.Flatten(fun p t -> List.SetNth p (tm.m t) (tm.c t)))
                 pr.zero,
        fun c m -> List.SetNth pr.zero m c)
```

6.1.3 Module Specializations for Gaussian Elimination

Using the algebra of linear polynomials defined in §6.1.2 and properties learned in §6.1.1, we can specialize the modules use in our software to generate a new variation of Buchberger's algorithm. This specialization will not only have the $\mathcal{O}(n^3)$ performance of Theorem 6.8, but also closely resemble a hand-made implementation of the Gaussian Elimination such as that defined by Carette and Kiselyov [16].

First, we define a function that implements Lemma 6.3 and decides which pairs of linear polynomials are "compatible". i.e., their residual (S-polynomial) is determined to be non-zero:

```
let Compatible (A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>)
               (c,i,j) = codegen {
    let! c_i = C.Get c i
    let! c_j = C.Get c j
    let! lm_i = A.LM c_i
    let! lm_j = A.LM c_j
    return (i ^<> j) ^&& (lm_i ^= lm_j)
}
```

The residual of polynomial p against polynomial q on element i is the polynomial $p - d \cdot q$ where $d = p_i/q_i$ whenever q_i is not zero. When p and q are compatible polynomials (according to Lemma 6.3 and the function `Compatible` above) and i is the index of the leading term of p and q, then the residual of p against q on element i is the S-polynomial of p and q as shown by Lemma 6.5.

The following code generator defines the function `residual` which performs this operation:

```
let GetResidual(A:PolynomialAlgebra<_,_,_,_>) =
    let res p q i =
        let lc1 = List.Nth p i
        let lc2 = List.Nth q i
        let t1 = A.scalar (A.CR.τ lc1 lc2) p
        let t2 = A.scalar (A.CR.τ lc2 lc1) q
        A.sub t1 t2
    DefineOnce "residual" (TernaryOp.Flatten res)
```

The first specialization of Buchberger's algorithm that we consider here is the implementation of Buchberger's Criteria 2.59 and 2.60 which only expands the working set by compatible pairs of polynomials within the same group P_i (as per Definition 6.2) and implements Lemmas 6.3 and 6.4:

```
let AddPairs(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>,S:WorkingSet<_,_,_>) =
    { new ExpansionStrategy<_,_,_,_> with
        member e.Init(c,s) = Prepend <| codegen {
            let! ids = C.Indexes c
            for p in Seq.AllPairs ids do
                let i,j = Fst p, Snd p
                let! comp = Compatible (A,C) (c,i,j)
                yield! IfU comp <| S.Add s (c,i,j)
        }
        member e.Expand _ (c,s,k) = codegen {
            let! ids = C.Indexes c
            for i in ids do
                let! comp = Compatible (A,C) (c,i,k)
                yield! IfU comp <| S.Add s (c,i,k)
        }
    }
```

Next, we implement Lemma 6.5 which defines a specialization of the S-polynomial computation using the residual function defined earlier:

```
let Residual(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>) =
    { new SPoly<_,_,_,_> with
        member p.Init _ = GetResidual A
        member p.σ res (c,i,j) = codegen {
            let! c_i = C.Get c i
            let! c_j = C.Get c j
            let! lm_i = LM A c_i
            return Function.Apply3 res c_i c_j lm_i
        }
    }
```

One major difference between Buchberger's Algorithm and Gaussian Elimination is that polynomial computations during the execution of Buchberger's Algorithm are added as new polynomials, whereas row operations in Gaussian Elimination are performed inline and do not introduce any new rows. In order to remove the superfluous polynomials left by Buchberger's algorithm, we need a specialization of the reduction strategy to reduce all such polynomials. Corollary 6.6 showed that we only have at most one polynomial from each group $P_0, \ldots, P_{n-1}$, therefore we may reduce any polynomial which is compatible with any other element of the basis:

```
let PruneRows(A:PolynomialAlgebra<_,_,_,_>,C:Container<_,_,_>) =
    { new ReductionStrategy<_,_,_> with
```

```
        member s.Init _ = Let <| Return (List.Empty())
        member s.Reduce rs (c,i) = codegen {
            let! red = Fun <| fun j -> codegen {
                let! comp = Compatible (A,C) (c,i,j)
                let seen = List.Contains rs j
                return (Not seen) ^&& comp
            }
            let! ids = C.Indexes c
            yield! If (Seq.Exists red ids)
                      (codegen {yield List.Add rs i; return True})
                      (Return False)
        }
    }
```

At this point, we know that the minimal Gröbner basis computed by these modules corresponds to the row-echelon reduced matrix that Gaussian Elimination produces. There are two more tasks that the canonicalization process needs to perform in order to produce a full implementation of Gaussian Elimination: back-propagation, and the reduction to "monic" polynomials. In this case, we define a monic polynomial to have the leading coefficient 1 (i.e., the first element of each row in matrix form is 1) whenever possible. When given a Field for coefficients, we may simply divide the polynomial by the leading coefficient to obtain a monic polynomial. In case of a quotient ring for coefficients, we only have access to a division-with-remainder operation, therefore we divide the polynomial by the gcd of all the coefficients (which is guaranteed to be divisible by all elements) to obtain the polynomial in its most reduced form:

```
let BackPropagate(A:PolynomialAlgebra<'a,_,_,_>,C:Container<_,_,_>) =
    { new CanonicalForm<_,_,_> with
        member p.Init _ = GetResidual A
        member f.Canonicalize r (c,p) = codegen {
            let! qs = C.All c

            // Back-propagate
            let! rem = Fun2 <| fun p q -> codegen {
                let! lm_p = LM A p
                let! lm_q = LM A q
                yield Control.If (lm_q ^> lm_p) (Function.Apply3 r p q lm_q) p
            }

            // Make the polynomials monic
```

```
        use cf = Seq.Fold rem p qs
        let! div = Fun <| fun c ->
            if A.CR :? Field<'a>
            then codegen {
                use q = A.LC cf
                return (A.CR :?> Field<'a>).div c q
                }
            else codegen {
                use q = List.Fold (BinaryOp.Flatten A.CR.gcd) (A.LC cf) cf
                return Option.UnOption(A.CR.div c q)
            }
        return List.Map div cf
    }
}
```

6.1.4 Generation of Gaussian Elimination Specialization

We may now generate a full specialization of Buchberger's Algorithm using the algebra of linear polynomials and the modules defined above by setting up the following specialization on field of rationals $\mathbb{Q}$ (using `BigRational` arithmetic):

```
let t = Debug.NoTrace
let bk = ReducedBasis
let K = Field.QQ
let pa = LinearPolynomial 4 K
let ip = Input.InBasis
let pc = Container.ListContainer
let ws = WorkingSet.DirectPick
let es = AddPairs(pa,pc,ws)
let sp = Residual(pa,pc)
let nr = NormalRemainder.NoRemainder
let rs = PruneRows(pa,pc)
let cf = BackPropagate(pa,pc)
let op = Output.OutBasis
let qge = GBSolver(t,bk,pa,ip,pc,ws,es,sp,nr,rs,cf,op)
let ge = GetV qge
```

The Gaussian Elimination algorithm that is generated by this specialization provides evidence of Objective 1.1(9) by showing that we are indeed able to specialize Buchberger's Algorithm to resemble a sub-algorithm textually and computationally:

```
val ge: seq<BigRational list> -> seq<BigRational list> = fun a_1 ->
```

```
let v_1 = new List<BigRational list> (a_1)
let v_2 = new List<int * int> ()
let lm i = List.findIndex (fun i -> 0N <> i) i
for i_1 in List.allPairs [0 .. v_1.Count - 1] do
    if (fst i_1) <> (snd i_1) && (lm v_1.[fst i_1]) = (lm v_1.[snd i_1]) then
        v_2.Add (fst i_1, snd i_1)
let residual i j k =
    List.map2 (fun i j -> i - j)
              (List.map (fun i -> j.[k] * i) i)
              (List.map (fun j -> i.[k] * j) j)
while v_2.Count > 0 do
    let t_1 = v_2.[0]
    v_2.RemoveAt 0
    let t_2 = residual v_1.[fst t_1] v_1.[snd t_1] (lm v_1.[fst t_1])
    if not (List.forall (fun i -> 0N = i) t_2) then
        let t_3 = v_1.Add t_2; v_1.Count - 1
        for i_1 in 0 .. v_1.Count - 1 do
            if i_1 <> t_3 && (lm v_1.[i_1]) = (lm v_1.[t_3]) then
                v_2.Add (i_1, t_3)
let v_3 = new List<int> ()
Seq.map (fun i_2 ->
        let t_1 =
            Seq.fold (fun a_3 a_4 ->
                    if (lm a_4) > (lm a_3)
                    then residual a_3 a_4 (lm a_4)
                    else a_3)
                v_1.[i_2]
                v_1
        let t_2 = List.find (fun i -> 0N <> i) t_1
        List.map (fun i -> i / t_2) t_1)
    (Seq.filter (fun i_1 ->
            let t_1 =
                if Seq.exists (fun a_2 ->
                        not (v_3.Contains a_2) &&
                        i_1 <> a_2 &&
                        (lm v_1.[i_1]) = (lm v_1.[a_2]))
                    {0 .. v_1.Count - 1}
                then v_3.Add (i_1); true
                else false
            not t_1)
        {0 .. v_1.Count - 1})
```

Example 6.9 We will revisit Example 6.7 using the algorithm generated above — with the minor modification of generating traces — to show that `ge` A_0 produces A_3'':

```
let A0 =
       [[2N;-1N;-1N;-1N;-1N];
        [1N;-1N;-1N; 0N; 0N];
        [0N; 3N;-2N; 1N;-4N];
        [1N; 3N; 0N;-2N;-3N]];;

> ge A0;;
Begin initialization
Initialization complete
Picked pair: (0, 1)
 - Residual = [0N; 1N; 1N; -1N; -1N]
 - Normalized = [0N; 1N; 1N; -1N; -1N]
 * Adding new polynomial #4
⋮
Begin post-processing
Reductions initiated
Returning result
Polynomial #0 is redundant
Polynomial #1 is redundant
Polynomial #2 is redundant
Polynomial #3 does not reduce
 - Canonicalized = [1N; 0N; 0N; 0N; -5N]
⋮
val it : seq<BigRational list> =
  seq
    [[1N; 0N; 0N; 0N; -5N];
     [0N; 1N; 0N; 0N; -2N];
     [0N; 0N; 1N; 0N; -3N];
     [0N; 0N; 0N; 1N; -4N]]
```

□

6.1.5 Fraction-Free Gaussian Elimination

Earlier in the definition of the `residual` function, we scaled the two polynomials by the crossfactors of p_i and q_i before subtracting; i.e., we computed $\tau_{p_i,q_i} \cdot p - \tau q_i, p_i \cdot q$ instead of $p - p_i/q_i \cdot q$ when resolving polynomial p against polynomial q on element i. These two polynomials are multiples of each other and may be used interchangeably

in the row reductions. When the coefficients of the polynomials form a field, both of these residuals are reduced to the exact same polynomial when they are scaled to the monic form during the canonicalization process.

On the other hand, when the coefficients only form a quotient ring, we witness a crucial difference between the two residuals above: The former is a residual of the two polynomials with the ith element made zero, but the latter may not exist due to existence of the remainders. Computing the residuals using the former formula allows us to perform Gaussian Elimination on quotient rings that only define the Euclidean algorithm for division with remainder and do not form a field.

This is the first step towards performing fraction-free Gaussian Elimination on rings. Notice that we have not defined any specializations of the working set in this chapter. The `Pick` function of the working set directly corresponds to the choice of pivoting strategy in Gaussian Elimination. We will only use the simple selection of pivots using `DirectPick` specialization of the working set, but the reader must know that a full implementation of fraction-free GE requires implementation of pivoting strategies as well.

We also specialized the computation of monic polynomials in the canonicalization module such that we have alternate method of simplifying polynomials when a full field division is not present.

Example 6.10 (Gaussian Elimination on Quotient Rings) Let us revisit Example 6.9 using the (limited) ring of machine integers. The only difference in generation of this example is the setup:

```
let K = QuotientRing.ZI
```

when generating the polynomial algebra. The resulting program has the signature:

```
val ge : (seq<int list> -> seq<int list>)
```

when fully generated. The following transcript demonstrates the Gaussian Elimination using `int` types:

```
> ge
      [[2;-1;-1;-1;-1];
       [1;-1;-1; 0; 0];
       [0; 3;-2; 1;-4];
       [1; 3; 0;-2;-3]];;
val it : seq<int list> =
  seq
```

```
[[-1; 0; 0; 0; 5];
 [0; 1; 0; 0; -2];
 [0; 0; 1; 0; -3];
 [0; 0; 0; 1; -4]]
```

□

6.2 Euclidean Algorithm

We will follow the same methodology as §6.1 to specialize an instance of Buchberger's Algorithm and generate the Euclidean algorithm. We know that in the case of univariate polynomials (i.e. polynomials of only one variable) then the Gröbner basis of a set of polynomials is the singleton set of their greatest common divisor [13]. In this section we will demonstrate this process by generating a program that implements precisely the Euclidean Algorithm for computing the polynomial gcd of a pair of polynomials.

6.2.1 Univariate Polynomial Algebra

A *univariate* polynomial $\rho \in \mathcal{R}[\vec{x}]$ is a special type of polynomial that utilizes only one variable. i.e., $|\vec{x}| = 1$. We will refer to this one variable simply by x.

Definition 6.11 (Representation of Univariate Polynomials) Let $\mathcal{R}$ be a quotient ring. Given a univariate polynomial $\rho \in \mathcal{R}[x]$, we represent ρ by the list $p = [p_0; \ldots; p_k]$ where $k = \deg(\rho)$, $\rho = p_0 + \sum_{i=1}^{k} p_i \cdot x^i$ and $p_k \neq 0_{\mathcal{R}}$.

Using term ordering $\sigma = \texttt{lex}$ we have $x^i >_\sigma x^j \Leftrightarrow i > j$ and thus for list representations p_1 and p_2, $p_1 >_\sigma p_2 \Leftrightarrow \texttt{len}(p_1) > \texttt{len}(p_2)$ ($\equiv \deg(\rho_1) > \deg(\rho_2)$, since $\texttt{len}(p) = \deg(\rho) + 1$ for all p).

The last element of the polynomial representation signifies the leading coefficient of the polynomial, i.e., $\mathrm{LC}_\sigma \rho = p_k$ and thus $\mathrm{LT}_\sigma \rho = p_k \cdot x^k$.

Definition 6.12 (Division with Remainder) Given two univariate polynomials $p, g \in \mathcal{R}[x]$, define the polynomials $q, r \in \mathcal{R}[x]$ (called the *quotient* and *remainder*, respectively) to be the result of Euclidean division of p by g such that $p = qg + r$ and either $\deg(r) < \deg(g)$ or that $\mathrm{LC}_\sigma r \not| \mathrm{LC}_\sigma g$.

Notice that this is a slightly modified version of the Euclidean long division algorithm to adjust for allowing of quotient rings instead of requiring a field for coefficients.

Similar to the monomial monoid defined in §6.1.2, we define the monomials for the algebra of univariate polynomials to be integer indexes such that for any $i \in \mathbb{N}$, i represents the monomial x^i. This implicitly means that the index 0 is the constant monomial:

```
let Monomials =
    MonomialMonoid.Gen(
        Idx.zero,
        Idx.add,
        id,
        Seq.Singleton,
        (fun p -> Seq.Singleton (Pair Idx.zero p)),
        Seq.Head,
        (fun p -> Snd (Seq.Head p)),
        (fun x y -> let d = Idx.sub x y
                    Control.If (d ^>= Idx.zero) (Option.Some d) (V None)),
        Min,
        Max,
        (fun x y -> Idx.sub (Max x y) x))
```

In order to perform computations on this representation of univariate polynomials, we need a special version of the `map2` function that performs the mapping operation on lists with different lengths. The list representation of polynomials is "padded" with enough zeros at the end to perform the operation, and then finally trimmed such that the length of the list is equivalent to the degree of the polynomial it represents:

```
let SpecialMap2 f z l1 l2 =
    let rec i_map2 = function
    | [],[] -> []
    | [],h::t -> f z h :: i_map2([],t)
    | h::t,[] -> f h z :: i_map2(t,[])
    | h1::t1,h2::t2 -> f h1 h2 :: i_map2(t1,t2)
    let rec trim = function
    | [] -> []
    | h::t when h=z -> trim t
    | l -> l
    i_map2(l1,l2) |> List.rev |> trim |> List.rev

let map2 f z l1 l2 =
    match f,z,l1,l2 with
```

```
    | V f,V z,V l1,V l2 -> SpecialMap2 f z l1 l2 |> V
    | _ -> E <@ SpecialMap2 (%GetE f) (%GetE z) (%GetE l1) (%GetE l2) @>
```

And finally, we define the algebra of univariate polynomials as follows:

```
let PolyRing (cr:QuotientRing<'a>) =
    let zero = V []
    let one = List.Singleton cr.one
    let neg = List.Map (UnaryOp.Flatten cr.neg)
    let add = map2 (BinaryOp.Flatten cr.add) cr.zero
    let sub = map2 (BinaryOp.Flatten cr.sub) cr.zero
    let pad i = List.Append (List.Replicate i cr.zero)
    let scale s = List.Map (UnaryOp.Flatten(cr.mul s))
    let mul x y =
        let parts = List.Mapi (BinaryOp.Flatten(fun i s -> pad i (scale s y))) x
        List.Fold (BinaryOp.Flatten add) zero parts
    let div p1 p2 =
        let l1,l2 = List.Length p1, List.Length p2
        let c1 = List.Nth p1 (Idx.sub l1 Idx.one)
        let c2 = List.Nth p2 (Idx.sub l2 Idx.one)
        let mkDiv d = Option.Some(pad (Idx.sub l1 l2) (List.Singleton d))
        if cr :? Field<'a> then
            let d = (cr :?> Field<'a>).div c1 c2
            Control.If (l1 ^> l2) (mkDiv d) (V None)
        else
            let d = cr.div c1 c2
            Control.If ((l1 ^> l2) ^&& (Option.IsSome d))
                       (mkDiv(Option.UnOption d)) (V None)
    let rem p1 p2 =
        let l1,l2 = List.Length p1, List.Length p2
        let c1 = List.Nth p1 (Idx.sub l1 Idx.one)
        let c2 = List.Nth p2 (Idx.sub l2 Idx.one)
        let s =
            if cr :? Field<'a> then
                let d = (cr :?> Field<'a>).div c1 c2
                fun x y -> cr.sub x (cr.mul y d)
            else
                fun x y -> cr.sub (cr.mul x c2) (cr.mul y c1)
        map2 (BinaryOp.Flatten s) cr.zero p1 (pad (Idx.sub l1 l2) p2)

    QuotientRing.Gen(zero, one, neg, add, mul, Some sub,
        None, None, Some div, Some rem, None, None, None)
```

```
let UnivariatePolynomial cr =
    let tm = TermModule.Generic(cr, Monomials)
    let pr = PolyRing cr
    let mk c m = List.Append (List.Replicate m cr.zero) (List.Singleton c)
    PolynomialAlgebra.Gen(
        cr, tm, pr,
        List.Singleton,
        Some(fun s -> List.Map (UnaryOp.Flatten (cr.mul s))),
        Compare,
        (fun p -> Seq.Mapi (BinaryOp.Flatten(fun i n -> tm.make n i)) p),
        Some List.Length,
        List.IsEmpty,
        (fun p -> let last = Idx.sub (List.Length p) Idx.one
                  tm.make (List.Nth p last) last),
        Some(fun p -> Idx.sub (List.Length p) Idx.one),
        Some(fun p -> List.Nth p (Idx.sub (List.Length p) Idx.one)),
        List.Reverse >> List.Tail >> List.Reverse,
        (fun ts ->
            let ps = Seq.Map (UnaryOp.Flatten(fun t -> mk (tm.c t) (tm.m t))) ts
            Seq.Fold (BinaryOp.Flatten pr.add) pr.zero ps),
        mk)
```

6.2.2 Euclidean Algorithm Specialization

The specializations of the modules for Euclidean Algorithm rely on the following (informal) specifications:

(1) All the polynomials are univariate.

(2) There are only two input values.

(3) At any point during the Euclidean Algorithm, we only need to store two elements.

(4) For polynomials $p_1, p_2 \in \mathcal{R}[x]$ such that $p_1 \geq_\sigma p_2$, we have $\mathbf{S}_{p_1,p_2} = \mathtt{rem}(p_1, p_2)$.

(5) No reduction or canonicalization step is needed.

In order to produce a program that closely resembles the statement of the Euclidean Algorithm, and to demonstrate the usage and flexibility of the input and output modules, we will implement a special instance of the `Container` module that

only stores two polynomials as separate variables and does not expand. The references to the two variables will be passed along through the generation as state records called `p1` and `p2`. This is essentially performing call-by-name on the two storage variables:

```
let PairContainer(A:PolynomialAlgebra<_,_,_,_>) =
    let s1,s2 = StateRecord("p1"),StateRecord("p2")
    { new Container<_,_,_> with
        member lc.Init s = Prepend <| codegen {
            let! p1 = s1.Lookup()
            let! p2 = s2.Lookup()
            return! IfU (A.deg (Deref p2) ^> A.deg (Deref p1)) (codegen {
                use t = Deref p1
                yield Assign p1 (Deref p2)
                yield Assign p2 t
            })
        }
        member lc.Add _ p = codegen {
            let! p1 = s1.Lookup()
            let! p2 = s2.Lookup()
            return! If (A.deg p ^> A.deg (Deref p2))
                (codegen {
                    yield Assign p1 p
                })
                (codegen {
                    yield Assign p1 (Deref p2)
                    yield Assign p2 p
                })
            return Unit
        }
        member lc.Get _ _ = Return A.zero
        member lc.All _ = Return <| Seq.Empty()
        member lc.Indexes _ = Return <| Seq.Empty()
    }
```

We also specialize the `Input` module to accept a tuple (pair) of polynomials as the input to the algorithm and define the two variables for storing as `p1` and `p2`. Similarly, the specialization of the `Output` module only returns the single value of the last non-zero remainder computed by the Euclidean Algorithm:

```
let PairInput<'p> =
    let s1,s2 = StateRecord("p1"),StateRecord("p2")
    { new Input<'p,'p*'p> with
        member i.Process p = codegen {
```

```
            use p1 = Ref (Fst p)
            use p2 = Ref (Snd p)
            do! s1.Extend p1
            do! s2.Extend p2
            return Seq.Empty()
        }
    }

let LastOutput<'p> =
    let s1,s2 = StateRecord("p1"),StateRecord("p2")
    { new Output<'p,'p> with
        member o.Process _ = codegen {
            let! p2 = s2.Lookup()
            return Deref p2
        }
    }
```

Using this strategy of storing two polynomials, we have a simplified instance of `WorkingSet` which always chooses the two aforementioned items as the critical pair. We will define the working set itself as a Boolean flag which signals when a final remainder (the zero polynomial) is obtained and terminates the loop. The updating of this flag is performed by the expansion strategy:

```
let PairPick =
    let s1,s2 = StateRecord("p1"),StateRecord("p2")
    { new WorkingSet<_,_,_> with
        member s.Init _ = Let <| codegen {
            return Ref True
        }
        member s.Add w (_,_,_) = codegen {
            return Assign w True
        }
        member s.Del _ (_,_,_) = Return Unit
        member s.HasMore w _ = codegen {
            return Deref w
        }
        member s.Pick w _ = codegen {
            yield Assign w False
            let! p1 = s1.Lookup()
            let! p2 = s2.Lookup()
            return Pair Unit Unit
        }
```

```
        member s.All _ _ = Return <| Seq.Empty()
}

let FlagExpansion (S:WorkingSet<_,_,_>) =
    { new ExpansionStrategy<_,_,_,_> with
        member e.Init(_,_) = Return Unit
        member e.Expand _ (c,s,i) = S.Add s (c,i,i)
    }
```

Finally, the specialization of the S-Polynomial module simply computes the remainder of the two polynomials in the state and returns the value:

```
let Remainder(A:PolynomialAlgebra<_,_,_,_>) =
    let s1,s2 = StateRecord("p1"),StateRecord("p2")
    { new SPoly<_,_,_,_> with
        member p.Init _ = Return Unit
        member p.σ _ (_,_,_) = codegen {
            let! p1 = s1.Lookup()
            let! p2 = s2.Lookup()
            return A.rem (Deref p1) (Deref p2)
        }
    }
```

6.2.3 Generation of Euclidean Algorithm

Using the polynomial algebra of univariate polynomials and the module specializations of Euclidean Algorithm, we generate our instance of Buchberger's algorithm by the following setup using the field of rationals $\mathbb{Q}$ (`BigRational` arithmetic) for coefficients:

```
let t = Debug.NoTrace
let bk = StandardBasis
let K = Field.QQ
let pa = UnivariatePolynomial K
let ip = PairInput
let pc = PairContainer pa
let ws = PairPick
let es = FlagExpansion ws
let sp = Remainder pa
let nr = NormalRemainder.NoRemainder
let rs = ReductionStrategy.NoReduction
let cf = CanonicalForm.NoOperation
```

```
let op = LastOutput
let qea = GBSolver(t,bk,pa,ip,pc,ws,es,sp,nr,rs,cf,op)
let ea = GetV qea
```

And we obtain the following generated code as the Euclidean specialization of Buchberger's algorithm. Notice that the signature of the generated function has changed due to the specialization of the input and output modules:

```
val ea: BigRational list * BigRational list -> BigRational list = fun a_1 ->
    let t_1 = ref (fst a_1)
    let t_2 = ref (snd a_1)
    if List.length(!t_2) > List.length(!t_1) then
        let t_3 = !t_1
        t_1 := !t_2
        t_2 := t_3
    let v_1 = ref true
    while !v_1 do
        v_1 := false
        let t_3 =
            SpecialMap2 (fun i j ->
                   i - (j * ((!t_1).[List.length(!t_1) - 1] /
                             (!t_2).[List.length(!t_2) - 1])))
                ON
                !t_1
                (List.replicate (List.length(!t_1) - List.length(!t_2)) ON) @ (!t_2)
        if not t_3.IsEmpty then
            if List.length t_3 > List.length(!t_2)
            then t_1 := t_3
            else
                t_1 := !t_2
                t_2 := t_3
            v_1 := true
    !t_2
```

□

Chapter 7

Conclusion

7.1 Achieved Goals

Back in the introduction chapter we defined the goal of this thesis: to generate specialized instances of Buchberger's algorithm that produce the solutions to a great variety of computational commutative algebra problems by converting the problem into a question of finding (minimal, reduced) Gröbner bases and carefully tailoring every aspect of the algorithm to match the properties of the question at hand. At this point, with the final results achieved in §5, we have demonstrated that we can generate thousands of different variations and flavours of Buchberger's algorithm designed for each individual task.

In §1.1 we also defined seven immediate objectives that we achieved in this thesis. We described the mathematical background required for Gröbner bases in §2. In §3 we designed the framework for code generation in F#. We formalized software specification and specialization in §4. §5 designed the architecture for generating Gröbner bases solvers, defined the computational algebra library in F#, and developed the code generator for Buchberger's algorithm. Finally, we generated some sample specializations of sub-algorithms of Buchberger's algorithm to demonstrate the usefulness of this code generator.

This concludes the objectives and the components of this research that we had aimed to accomplish before starting this research.

We had also defined four long-term objectives to be partially answered by this thesis. We believe that with the samples and tutorials in §5.3.3 we have demonstrated the usefulness of software specialization and code generation techniques by, not only

generating different use cases for the algorithm, but also by showing the applications of this generator in teaching and understanding of Gröbner bases. We also showed that the generated instances of Buchberger's algorithm can, in fact, directly resemble the algorithms for special-domain problems.

Additionally, we configured the framework and provided some simple proofs to show correctness of the generated algorithms, but we did not generate any of the proofs in this thesis, and there is still much more work to be done to achieve this goal. We also attempted to generate the specific instances of this algorithm that resembles Gaussian Elimination in §6, but did not fully produce all the specialized instances.

7.2 Summary of Results

We will revisit all the material covered in this thesis and summarize the objectives that we achieved:

(1) §2.1 recaps all the basic material in abstract algebra and polynomial algebras. It also defines important terminology about monomials and polynomial ordering.

(2) §2.2 defines the computational aspect of polynomial evaluation and introduces three classes of problems that naturally arise from performing polynomial computations: solving systems of equations (§2.2.2) and their relation to algebraic geometry, ideal membership problems (§2.2.3) with a focus on the polynomial division algorithm, and polynomial rewrite systems (S2.2.4) with an emphasis on confluence.

(3) §2.2.5 discusses Gröbner bases as a viable solution to the problems outlined earlier and introduces related concepts such as minimal and reduced Gröbner basis. It also states some of the important properties of Gröbner bases, such as the uniqueness of reduced bases. Finally, §2.2.6 defines the important theorems regarding elimination and extension of ideals for solving systems of equations.

(4) §2.3 describes a version of Buchberger's algorithm for finding Gröbner bases using S-polynomials and normal forms, and then gives the algorithms for computing minimal and reduced bases in §2.3.3.

(5) §2.3.4 shows some important optimizations for Buchberger's algorithm and discusses the execution time and space complexity of the algorithm, and then states an improved version of Buchberger's algorithm based on these observations.

(6) §2.3.5 provides a step-by-step tutorial on how to compute a reduced Gröbner basis for a sample set.

(7) §2.4 defines how to derive other related computational algorithms from Buchberger's algorithm and provides references for these transformations.

(8) §3.1 introduces the meta programming elements of the F# language.

(9) §3.1.2 defines a formal framework for reasoning about the syntax of F# and formally defines quasiquotation and splicing. This framework also leads to a research paper [31] that Dr. William Farmer and I are working on for publication.

(10) §3.1.3 extends the F# compiler and libraries to compile nested quasiquotations and mixed-stage variables within quoted code. We also provide the proof of the correctness theorems for this extension and a brief overview of the inner workings of the library.

(11) §3.1.3 also provides a new pretty printer for F# quotations that we use purely for presentation purposes in this thesis.

(12) §3.2 programs the combinators for generating code in F# using continuation-passing style as described in §3.2.1.

(13) §3.2.2 creates a flexible and extensible state for this code generator that can track heterogeneous-typed data in one container and is passed by every codegen combinator.

(14) §3.2.4 defines a domain-specific language to generate code using the "computation expressions" featured in the F# programming language.

(15) §4.1.1 describes a formal framework for writing specifications for software products based on higher-order logic that can specify both values as invariants, and program specifications as pre/post-conditions (covariants and contravariants).

(16) §4.1.2 provides the formal definitions for the terms refinement, actualization and specialization in this framework, as well as the definitions for the companion

terms abstraction, conceptualization and generalization as opposite actions to the former concepts.

(17) §4.2 defines two approaches to writing modules — feature-oriented modules and aspect-oriented modules — and their advantages and disadvantages in different scenarios. We also describe how we may use both types of modules in the same software architecture to take advantage of the benefits they provide.

(18) §4.2.2 creates a language for writing specifications for modules as collections of both invariants and methods. We also establish the required terminology for implementing modules and parametrizing module interfaces.

(19) §4.2.4 summarizes how to decompose a software architecture into a modular design using the concepts established by this paper and by earlier software engineering literature. This section also provides a demonstration of the modular breakdown by decomposing the architecture of a KWIC indexing system.

(20) §5.1.3 designs the module structure for an abstract algebra library in F# and specifies the interfaces for these modules to be used in computational algebra programs.

(21) §5.1.4 describes the methods of implementing each statement in the specifications of the algebra modules as computer programs in such a way that the implementation satisfies the algebraic interface as closely as possible.

(22) §5.1.4 also programs the generators for these algebra modules and implements the most common uses of some of the algebraic objects.

(23) §5.1.5 specifies the module interfaces for polynomial algebra objects — monomials, terms, term orderings, and polynomials — using the algebra libraries established earlier.

(24) §5.1.5 also programs the generators for polynomial algebra library and implements some of the most common approaches to monomials, terms, orderings and polynomials.

(25) §5.2 decomposes Buchberger's algorithm and the related minimal and reduced Gröbner bases algorithms into a modular design in order to create a program family of Buchberger-like algorithms.

(26) §5.3.1 specifies the module interfaces for the modules that were designed during the decomposition of Buchberger's algorithm.

(27) §5.3.2 programs the final code generator that generates specialized instances of Buchberger's algorithm by using the modules from the earlier breakdown of Buchberger's algorithm and the modules from the polynomial algebra library.

(28) §5.3.3 implements some basic and common versions of the modules used in Buchberger decomposition that provide a basis for any standard instance of this algorithm.

(29) Example 5.48 generates many sample instances of Buchberger's algorithm and compares the generated version of this algorithm as produced by the code generator with the pseudocode of Buchberger's algorithm which was stated earlier.

(30) Further in §5.3.3 we implement all the listed improvements to Buchberger's algorithm as specializations of the modules in this library and finally generate all the improved versions automatically. We draw a conclusion by demonstrating how many variations and flavours of Buchberger's algorithm we were able to generate using just the sample modules implemented in this chapter.

(31) Example 5.53 demonstrates how the tutorials of Gröbner bases computations may be generated by this program.

(32) §6 shows how more specializations of the Buchberger's algorithm modules can lead to vastly different algorithms that no longer resemble Gröbner bases directly, but are special sub-algorithms of Buchberger's algorithm as defined earlier in §2.4.

(33) §6.1 generates a program to perform Gaussian Elimination directly from the code generator for Buchberger's algorithm to demonstrate the extent of the generated algorithms.

(34) §6.2 generates a program to perform the Euclidean Algorithm on a pair of univariate polynomials by specializing Buchberger's algorithm.

7.3 Further Work

As the result of the last few long-term objectives that we had defined earlier, we would like to conclude this thesis by sketching a road map of some future work that can be done to improve the results and achieve the remaining goals. We will outline three immediate results that would extend this research: Providing micro specializations (alongside with the macro specializations defined in this thesis) to further improve the algorithm. Implementing some of the known extensions to the Buchberger's algorithm that provide greater range of results for the generated algorithm; in this case, we have chosen the construction of a transformation matrix to improve the algorithm. And finally, to formally produce the proof of correctness for any of these generated algorithms.

7.3.1 Micro Specializations

As Carette and Kiselyov [16] demonstrated in the case of Gaussian Elimination, there are many choices for micro specializations of the Buchberger sub-algorithms that can be performed to further improve the optimization and level of details in the result of the algorithm. The implementations of the aspect modules in the Buchberger decomposition could benefit from more generation parameters that further specialize them. For example, the modules defined in §6.1 could further be parametrized to directly reflect all the design criteria that were outlined in [16].

7.3.2 Transformation Matrix

The "Extended Buchberger Algorithm" as defined by Kreuzer [49] produces a transformation matrix that defines how to change the basis of any polynomials from the original basis to the the computed Gröbner basis: Given a set of polynomials $F = \{f_1, \ldots, f_n\} \subset \mathcal{R}[\vec{x}]$, compute a σ-Gröbner basis $G = \{g_1, \ldots, g_m\} \subset \mathcal{R}[\vec{x}]$ such that $m \geq n$ and $\langle F \rangle = \langle G \rangle$, together with an $n \times m$-matrix $A = (a_{ij})$, $a_{ij} \in \mathcal{R}[\vec{x}]$ for $1 \leq i \leq n$, $1 \leq j \leq m$, such that $g_j = a_{1j} f_1 + \cdots + a_{nj} f_n$ for $1 \leq j \leq m$.

Implementing the extended version of Buchberger's algorithm with a optional modules for carrying, computing, and returning this transformation matrix further improves the range of the generated algorithms.

7.3.3 Providing Correctness Proofs

Finally, the most ambitious future direction for this thesis is to properly provide the proof of correctness of the generated algorithms as generated machine proofs alongside with the program itself. We believe that theorem proving environments that integrate with .NET environment (such as Z3 [25]) are the optimal choice for the proof generator.

Bibliography

[1] "Polymorphic data types, objects, modules and functors: is it too much?," in *RR 014, LIP6, UNIVERSITÉ PARIS 6*, 2000.

[2] "DSL implementation in MetaOCaml, Template Haskell, and C++," in *LNCS Volume 3016*, pp. 51–72, Springer-Verlag, 2004. http://www.cs.rice.edu/~taha/publications/journal/dspg04b.pdf.

[3] J. Abbott, A. Bigatti, M. Caboara, and L. Robbiano, "CoCoA: Computations in Commutative Algebra," *SIGSAM Communications in Computer Algebra*, 2007. also presented as Software Demo of the ISSAC07 Conference.

[4] P. B. Andrews, *An introduction to mathematical logic and type theory: to truth through proof*. Applied Logic Series, Springer, second ed., 2002.

[5] S. Apel, T. Leich, and G. Saake, "Aspectual mixin layers: Aspects and features in concert," in *In Proc. of Intl. Conf. on Software Engineering*, pp. 122–131, ACM Press, 2006.

[6] R. J. Back and J. Wright, *Refinement Calculus: A Systematic Introduction*. Springer, April 1998.

[7] D. Batory, "The design and implementation of hierarchical software systems with reusable components," *ACM Transactions on Software Engineering and Methodology*, vol. 1, pp. 355–398, 1992.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 30, no. 6, p. 2004, 2004.

[9] A. B. Bawden, "Quasiquotation in Lisp," in *O. Danvy, Ed., University of Aarhus, Dept. of Computer Science*, pp. 88–99, 1999.

[10] D. Box and A. Hejlsberg, "LINQ: .NET Language Integrated Query," 2010. http://msdn.microsoft.com/en-us/library/bb397926.aspx.

[11] B. Buchberger, "Gröbner bases: An algorithmic method in polynomial ideal theory," *Recent Trends in Multidimensional Systems Theory*, 1985.

[12] B. Buchberger, *An Algorithm for Finding the Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal*. PhD thesis, University of Innsbruck Institute of Mathematics, 1965.

[13] B. Buchberger and F. Winkler, *Gröbner Bases and Applications*. No. 251 in London Mathematical Society Lecture Nore Series, Cambridge University Press, 1998.

[14] J. Carette, "Gaussian Elimination: A case study in efficient genericity with MetaOCaml," *Science of Computer Programming*, vol. 62, no. 1, pp. 3–24, 2006.

[15] J. Carette, M. Elsheikh, and S. Smith, "A generative geometric kernel," in *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, (New York, NY, USA), pp. 53–62, ACM, 2011.

[16] J. Carette and O. Kiselyov, "Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code," *Science of Computer Programming*, vol. 76, no. 5, pp. 349–375, May 2011.

[17] J. Carette, O. Kiselyov, and C. C. Shan, "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages," *J. Funct. Program.*, vol. 19, no. 5, pp. 509–543, Sept. 2009.

[18] Y. Chang and B. W. Wah, "Polynomial Programming Using Groebner Bases," *Computer Software and Applications Conference*, 1994.

[19] A. Colyer, A. Rashid, and G. Blair, "On the separation of concerns in program families," tech. rep., 2004.

[20] C. Consel, "From a Program Family to a Domain-Specific Language," in *Domain-Specific Program Generation*, pp. 19–29, Springer Berlin Heidelberg, 2004.

[21] C. Consel and R. Marlet, "Architecturing software using a methodology for language development," in *Proceedings of the 10 th International Symposium on Programming Language Implementation and Logic Programming, number 1490 in Lecture Notes in Computer Science*, pp. 170–194, 1998.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, second ed., 2001.

[23] D. A. Cox, "Gröbner bases tutorial." Department of Mathematics and Computer Science, Amherst College. ISSAC 2007 Tutorial.

[24] D. A. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3/e (Undergraduate Texts in Mathematics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[25] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[26] J. C. Dehnert and A. A. Stepanov, "Fundamentals of Generic Programming," in *Generic Programming*, pp. 1–11, 1998.

[27] E. W. Dijkstra, "On the role of scientific thought." revised as [28], Aug. 1974.

[28] E. W. Dijkstra, "On the role of scientific thought," in *Selected Writings on Computing: A Personal Perspective*, pp. 60–66, Springer-Verlag, 1982.

[29] D. Dummit and R. Foote, *Abstract algebra*. Prentice Hall, 1999.

[30] W. M. Farmer, "Modules for a large library of formalized mathematics," in *AMS Special Session on Formal Mathematics for Mathematicians: Developing Large Repositories of Advanced Mathematics*, 2011.

[31] W. M. Farmer and P. Larjani, "Frameworks for reasoning about syntax that utilize quotation and evaluation," tech. rep., McMaster University, 2011. preprint, 33 pp.

[32] W. M. Farmer, "Theory interpretation in simple type theory," in *Higher-order algebra, logic, and term rewriting, volume 816 OF lecture notes in computer science*, pp. 96–123, Springer-Verlag, 1993.

[33] W. M. Farmer, "The seven virtues of simple type theory," *Journal of Applied Logic*, vol. 6, no. 3, pp. 267–286, Sept. 2008.

[34] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "Little theories," in *Automated Deduction | CADE-11, volume 607 of Lecture Notes in Computer Science*, pp. 567–581, Springer-Verlag, 1992.

[35] J.-C. Faugère, "A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)," *Journal of Pure and Applied Algebra*, vol. 139, no. (1–3), pp. 61–88, June 1999.

[36] J.-C. Faugère and S. Lachartre, "Parallel Gaussian Elimination for Gröbner bases computations in finite fields," in *ACM proceedings of The International Workshop on Parallel and Symbolic Computation (PASCO*, pp. 1–10, ACM, 2010.

[37] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, (New York, NY, USA), pp. 237–247, ACM, 1993.

[38] D. Garlan, "Architectures for software systems." Carnegie Mellon University.

[39] R. Gebauer and H. M. Möller, "On an installation of buchberger's algorithm," *J. Symb. Comput.*, vol. 6, no. 2-3, pp. 275–286, Dec. 1988.

[40] J. Harrop, *F# for Scientists*. New York, NY, USA: Wiley-Interscience, 2008.

[41] C. J. Hillar and T. Windfeldt, "Algebraic characterization of uniquely vertex colorable graphs," *J. Comb. Theory Ser. B*, vol. 98, no. 2, pp. 400–414, 2008.

[42] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.

[43] S. Hosten and R. Thomas, "Gröbner bases and integer programming."

[44] INRIA, "Objective caml." http://caml.inria.fr/ocaml/.

[45] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings of the 27th international conference on Software engineering*, ICSE '05, (New York, NY, USA), pp. 49–58, ACM, 2005.

[46] J. W. Klop, *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2003.

[47] D. Knuth and P. Bendix, "Simple word problems in universal algebras," *Computational Problems in Abstract Algebra*, pp. 263–297, 1970.

[48] M. Kreuzer and L. Robbiano, *Computational Commutative Algebra 1*. Springer, 2000.

[49] M. Kreuzer and L. Robbiano, *Computational commutative Algebra 2*. Springer, 2005.

[50] B. Kutzler and S. Stifter, "Automated geometry theorem proving using buchberger's algorithm," in *SYMSAC '86: Proceedings of the fifth ACM symposium on Symbolic and algebraic computation*, (New York, NY, USA), pp. 209–214, ACM, 1986.

[51] J. A. D. Loera, J. Lee, S. Margulies, and S. Onn, "Expressing combinatorial optimization problems by systems of polynomial equations and the nullstellensatz," *RC24276*, 2007. IBM Research Division.

[52] R. Marlet, S. Thibault, and C. Consel, "Efficient implementations of software architectures via partial evaluation," *Automated Software Engg.*, vol. 6, no. 4, pp. 411–440, 1999.

[53] MathScheme, "The MathScheme project." http://www.cas.mcmaster.ca/research/mathscheme/.

[54] E. W. Mayr, "Some complexity results for polynomial ideals," 1997.

[55] E. W. Mayr and A. R. Meyer, "The complexity of the word problems for commutative semigroups and polynomial ideals," *Advances in Mathematics*, vol. 46, no. 3, pp. 305 – 329, 1982.

[56] M. McGettrick, "Buchberger algorithm–gröbner basis–sparse multivariate polynomials." http://grobner.nuigalway.ie/grobner/basis.html.

[57] Microsoft Research, "The `F#` Power Pack," 2010. http://fsharppowerpack.codeplex.com/.

[58] A. Middeldorp and M. Starcevic, "A rewrite approach to polynomial ideal theory," tech. rep., 1991.

[59] MLton team, "Property list." http://mlton.org/PropertyList.

[60] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, pp. 55–92, 1989.

[61] S. Moritsugu and C. Arai, "Geometry theorem proving by Gröbner bases: Using ideal decompositions," *ACM Commun. Comput. Algebra*, vol. 42, no. 3, pp. 158–159, 2008.

[62] MSDN, "Generics in the .NET frameworks," 2012.

[63] MSDN, "Microsoft `F#` developer center and language reference," 2012. http://msdn.microsoft.com/en-us/library/dd233181.aspx.

[64] D. R. Musser and A. A. Stepanov, "Algorithm-Oriented Generic Libraries," *Software Practice and Experience*, vol. 24, pp. 623–642, 1994.

[65] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, pp. 1053–1058, 1972.

[66] D. L. Parnas, "On the design and development of program families," *IEEE Trans. Softw. Eng.*, vol. 2, no. 1, pp. 1–9, 1976.

[67] D. L. Parnas, "Designing software for ease of extension and contraction," in *Proceedings of the 3rd international conference on Software engineering*, ICSE '78, (Piscataway, NJ, USA), pp. 264–277, IEEE Press, 1978.

[68] D. L. Parnas, "Information distribution aspects of design methodology," in *IFIP Congress (1)*, pp. 339–344, 1971.

[69] G. O. Passmore and L. D. Moura, "Superfluous S-polynomials in Strategy-Independent Gröbner Bases," in *Symbolic and Numerical Algorithms for Scientific Computing*, pp. 45–53, 2009.

[70] T. Petricek and J. Skeet, *Real World Functional Programming: With Examples in F# and C#*. Greenwich, CT, USA: Manning Publications Co., 1st ed., 2009.

[71] J. C. Reynolds, "The discoveries of continuations," *Lisp Symb. Comput.*, vol. 6, no. 3-4, pp. 233–248, Nov. 1993.

[72] Rice University Programming Languages Team – PLT, "Metaocaml: A compiled, type-safe, multi-stage programming language." http://www.metaocaml.org/.

[73] T. Rompf, I. Maier, and M. Odersky, "Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform," *SIGPLAN Not.*, vol. 44, no. 9, pp. 317–328, Aug. 2009.

[74] S. Rugaber, "Software development process." Georgia Tech College of Computing.

[75] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[76] P. Schauenburg, "A Gröbner-based treatment of elimination theory for affine varieties," *J. Symb. Comput.*, vol. 42, no. 9, pp. 859–870, 2007.

[77] T. Sheard and S. P. Jones, "Template meta-programming for Haskell," *SIGPLAN Not.*, vol. 37, no. 12, pp. 60–75, 2002.

[78] C. M. U. Software Engineering Institute (SEI), "A framework for software product line practice," 2009.
http://www.sei.cmu.edu/productlines/index.html.

[79] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to C++," in *In Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002*, pp. 53–60, 2002.

[80] O. Spinczyk, D. Lohmann, and M. Urban, "Advances in AOP with AspectC++," 2003.

[81] B. Sturmfels, *Gröbner Bases and Convex Polytopes*. University Lecture Series, American Mathematical Society, 1996.

[82] Y. Sun and D. Wang, "A Generalized Criterion for Signature-based Algorithms to Compute Gröbner Bases," *CoRR*, vol. abs/1106.4918, 2011.

[83] G. J. Sussman and G. L. Steele, "Scheme: A interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, vol. 11, pp. 405–439, 1998. 10.1023/A:1010035624696.

[84] K. Swadi, W. Taha, O. Kiselyov, and E. Pasalic, "A monadic approach for avoiding code duplication when staging memoized functions," in *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, (New York, NY, USA), pp. 160–169, ACM, 2006.

[85] D. Syme and J. Margetson, "The `F#` Programming Language," 2011. http://fsharp.net/.

[86] D. Syme, "Leveraging .NET meta-programming components from `F#`: integrated queries and interoperable heterogeneous execution," in *ML '06: Proceedings of the 2006 workshop on ML*, (New York, NY, USA), pp. 43–54, ACM, 2006.

[87] D. Syme, G. Neverov, and J. Margetson, "Extensible pattern matching via a lightweight language extension," in *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, (New York, NY, USA), pp. 29–40, ACM, 2007.

[88] D. Syme, T. Petricek, and D. Lomov, "The `F#` asynchronous programming model," in *Proceedings of the 13th international conference on Practical aspects of declarative languages*, PADL'11, (Berlin, Heidelberg), pp. 175–189, Springer-Verlag, 2011.

[89] W. Taha, *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[90] W. Taha and T. Sheard, "MetaML and multi-stage programming with explicit annotations," *Theor. Comput. Sci.*, vol. 248, no. 1-2, pp. 211–242, 2000.

[91] Q. Tran and M. Y. Vardi, "Gröbner bases computation in Boolean rings for symbolic model checking," in *MOAS'07: Proceedings of the 18th conference on Proceedings of the 18th IASTED International Conference*, (Anaheim, CA, USA), pp. 440–445, ACTA Press, 2007.

[92] W. Trinks, "On Buchberger's method of solving systems of algebraic equations," *ACM Commun. Comput. Algebra*, vol. 45, no. 3/4, pp. 150–161, Jan. 2012.

[93] W. Van Orman Quine, *Mathematical Logic*. Harvard University Press, 1981.

[94] M. VanHilst and D. Notkin, "Decoupling change from design," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 6, pp. 58–69, 1996.

[95] D. Wang, *Gröbner Bases Applied to Geometric Theorem Proving and Discovering*. Cambridge University Press, 1998.

[96] V. Weispfenning and T. Becker, *Gröbner Bases: A Computational Approach to Commutative Algebra*, vol. 141 of *Series: Graduate Texts in Mathematics*. Springer-Verlag New York, Inc., 1998.

[97] Wikipedia, the free encyclopedia, "Gröbner basis." http://en.wikipedia.org/wiki/Gr%C3%B6bner_basis.

[98] J. Xu, *Mei A Module System for Mechanized Mathematics Systems*. PhD thesis, McMaster University, 2008.