Employing C++ Templates in the Design of a Computer Algebra Library

Alexander Brandt[1], Robert H.C. Moir[2], Marc Moreno Maza[3]

Department of Computer Science, The University of Western Ontario, London, Canada
[1] `abrandt5@uwo.ca`, [2] `rmoir3@uwo.ca`, [3] `moreno@csd.uwo.ca`

Abstract

We discuss design aspects of the open-source Basic Polynomial Algebra Subprograms (BPAS) library. We build on standard C++11 template mechanisms to improve ease of use and accessibility. The BPAS computer algebra library looks to enable end-users to do work more easily and efficiently through optimized C code wrapped in an object-oriented and user-friendly C++ interface. Two key aspects of this interface to be discussed are the encoding of the algebraic hierarchy as a class hierarchy and a mechanism to support the combination of algebraic types as a new type. Existing libraries, if encoding the algebraic hierarchy at all, use runtime value checks to determine if two elements belong to the same ring for an incorrect false sense of type safety in an otherwise statically-typed language. On the contrary, our template metaprogramming mechanism provides true compile-time type safety and compile-time code generation. The details of this mechanism are transparent to end-users, providing a very natural interface for an end-user mathematician.

Keywords: algebraic hierarchy · C++ templates · type safety

1 Introduction

In the world of computer algebra software there are two main categories. The first is computer algebra systems, self-contained environments providing an interactive user-interface and usually their own programming language. Custom interpreters and languages yield powerful functionality and expressibility, however, obstacles remain. For a basic user, they must learn yet another programming language. For an advanced user, interoperability and obtaining fine control of hardware resources is challenging. Axiom [12] is a classic example of such a system. Moreover, these problems are exacerbated by systems being proprietary and closed-source, such as Maple [5], Magma [6], and Mathematica [18]. The second category is computer algebra libraries, which add support for symbolic computation to an existing programming environment. Since such libraries extend existing environments, and are often free (as in free software), they can have a lower barrier to entry and better accessibility. Some examples are NTL [14], FLINT [11], and CoCoALib [1].

The Basic Polynomial Algebra Subprograms (BPAS) library [2] is a free and open-source computer algebra library for polynomial algebra, and is the subject of this paper. The BPAS library looks to improve the efficiency of end-users through both usability and performance, providing high-performance code along with an interface which incorporates some of the expressibility of a custom computer algebra system. The library's core is implemented in C for performance and wrapped in a

C++ interface for usability. Like any computer algebra software, functionality is highly important, yet usability makes the software practical.

The implementation of BPAS is focused on performance for modern computer architectures by optimizing for data locality and through the effective use of parallelization. These techniques have been applied to our implementations of multi-dimensional FFTs, real root isolation, dense modular polynomial arithmetic, and dense integer polynomial multiplications; see [7] and references therein. Recent works have extended BPAS to include arithmetic over large prime fields [8] and sparse multivariate polynomial arithmetic [3]. Experimentation presented in those works indicates that the performance of BPAS surpasses other existing works. All of this functionality culminates into a high-performance and parallel polynomial system solver (currently under development) based on the theory of regular chains [4]. However, in the present discussion, we look to describe our efforts to make these existing high-performance implementations accessible and practical through user-interface design and improved usability.

Usability includes many things: ease of use in interfaces, syntax, and semantics; mathematical correctness; accessibility and extensibility for end-users; and maintainability for developers. The BPAS library follows two driving principles in its design. The first is to encapsulate as much complexity as possible on the developer's side, where the developer's intimacy with the code allows her to bear such a burden, in order to leave the end-user's code as clean as possible. The second can be described by a common phrase in user experience design: "make it hard to do the wrong thing."

The object-oriented nature of C++, along with its automatic memory management, provides a very natural environment for a user-interface. While C++ is notoriously difficulty to learn, it remains ubiquitous in industry and scientific computing, making it reasonably accessible, and particularly so, if complexity can be well-encapsualted. Moreover, C++ being a compiled, statically- and strongly-typed language, further aids the end-user. The compilation process itself provides the user with checks on their code before it even runs. Meanwhile, statically-typed languages have been shown to be beneficial to usability, and decreases development time, compared to dynamic languages [10].

In the present work, we discuss our early efforts to use C++ metaprogramming to aid in the usability of our interface, for which we hope that BPAS will be easily adopted by other practitioners. Our discussion focuses on two aspects relating to type safety and expressibility. First, encoding the algebraic hierarchy as a class hierarchy is discussed in Section 2. Doing so while maintaining type safety is difficult; syntactically valid operations may yield mathematically invalid operations between incompatible rings. Secondly, we examine a mechanism to automatically adjust the definition of a class created from the composition of other classes. In particular, we look at polynomials adapting to different ground rings in Section 3. Our techniques are discussed and contrasted with existing works in Section 4. We conclude and present future work in Section 5.

We note that our techniques are not entirely new; the underlying template metaprogramming constructs have been adopted into the C++ standard since as early as C++11. Nevertheless, it remains useful to explore how these advanced concepts can be employed in the context of computer algebra. For details on C++, templates, and their capabilities, see [17].

2 Algebraic Hierarchy as a Class Hierarchy

In object-oriented programming (OOP) classes form a fundamental part of software design. A class defines a type and how all instances of that type should behave. Through a class hierarchy, or a tree of inheritance, classes have increasing specialization while maintaining all of the functionality of their superclasses. The benefits of a class hierarchy are numerous, including providing a common interface to which all objects should adhere, minimizing code duplication, facilitating incremental design, and of course, polymorphism. All of this provides better maintainability of the software and a more natural use of the classes themselves since they directly model their real-world counterparts.

For algebraic structures, the chain of class inclusions naturally admits an encoding as a class hierarchy. For example, the class inclusions of some rings[1],

$$\text{field} \subset \text{Euclidean domain} \subset \text{GCD domain} \subset \text{integral domain} \subset \text{ring},$$

would allow rings as the topmost superclass with an incremental design down to fields. Let us call such an encoding of algebraic types as a class hierarchy the *algebraic class hierarchy*. Particularly, we look to implement this hierarchy as a collection of abstract classes for the benefits of code re-use and enforcing a uniform interface across all concrete types (e.g. integers, rational numbers).

Unfortunately, an encoding of algebraic structures as classes in this way yields incorrect type safety. Through polymorphism, two objects sharing a superclass interact and behave in a uniform way, without regard to if they are mathematically compatible. Consider the C++ function declaration which could appear in the topmost `Ring` class: `Ring add(Ring x, Ring y)`. By polymorphism, any two Ring objects could be passed to this function to produce valid code, but, if those objects are from mathematically incompatible rings, this will certainly lead to errors. A more robust system is needed to facilitate strict type safety.

Some libraries (see Section 4) solve this by checking runtime values to ensure compatibility, throwing an error otherwise. Instead, our main idea is to define the interface of a ring (or a particular subclass, e.g. integral domain) in such a way where a function declaration itself restricts its parameters to be from compatible rings.

In our algebraic class hierarchy, function declarations themselves restrict their parameters to be from compatible rings through the use of template parameters. Particularly, our algebraic class hierarchy is a hierarchy of class templates with the template parameter `Derived`. This template parameter identifies the concrete ring(s) with which the one being defined is compatible. In this design, all abstract classes in the hierarchy have the template parameter `Derived` while the concrete classes instantiate this template parameter of their superclass with that concrete class itself being defined. This yields the C++ idiom, the Curiously Recurring Template Pattern (CRTP) (see [17, Ch. 16]).

While CRTP has several functions, it is used here to facilitate *static polymorphism*. That is to say, it forces function resolution to occur at compile-time, instead of dynamically at runtime via virtual tables, providing compile-time errors for incompatibility. For example, the topmost `Ring` class would become a class tem-

[1]Throughout this paper we assume commutative rings with unity.

plate `Ring<Derived>` and the `add` function would become `Derived add(Derived x, Derived y)`.

This process works from a key observation when considering simultaneously templates and class inheritance: different template parameter specializations produce distinct classes and thus distinct inheritance hierarchies. Recall that template instantiation in fact causes code generation at compile-time. Thus, each concrete ring defined via CRTP exists in its own class hierarchy, and dynamic dispatch via polymorphism cannot cause runtime inconsistencies. This concept is illustrated in Listing 1 where the abstract classes for ring and Euclidean domain are shown, as well as the concrete class for the ring of integers. The `Integer` class uses template instantiation where it defines its superclass, specializing the `Derived` parameter of `BPASEuclideanDomain` to be `Integer`, following CRTP.

```
template <class Derived>
class BPASRing;

//... more abstract algebraic classes, e.g. BPASGCDDomain, BPASField

template <class Derived>
class BPASEuclideanDomain : BPASGCDDomain<Derived>;

class Integer : BPASEuclideanDomain<Integer>;
```

Listing 1: A subset of the algebraic class hierarchy, using CRTP to declare the integers.

While this design provides the desired compile-time type safety, it may be viewed as too strict, since each concrete ring exists in an independent class hierarchy. For example, arithmetic between integers and rational numbers would be restricted. More generally, natural ring embeddings are neglected. However, we can make use of *implicit conversion* in C++. Where a constructor exists for type `A` taking an object of type `B` as input, an object of type `B` can be implicitly converted to an object of type `A`, and used anywhere type `A` is expected. A `RationalNumber` constructor taking an `Integer` parameter thus allows for automatic and implicit conversion, allowing integers to be used as rational numbers.

This design via implicit conversion can be seen as giving permission for compatibility between rings by defining such a constructor. Errors are then discovered at compile-time where implicit conversion fails. This is in opposition to other methods which act in a restrictive manner, allowing everything at compile-time and then throwing errors at runtime if incompatible.

We now look to extend the abstract algebraic class hierarchy to include polynomials. For genericity and a common structured interface we wish to parameterize polynomials by their ground ring. This can be accomplished with a secondary template parameter in addition to the `Derived` parameter already included by virtue of polynomials existing in the algebraic class hierarchy (see Listing 2).

However, this is not fully sufficient, and two issues arise. First, while polynomials do form a ring, they often form more specialized algebraic structures, e.g. a GCD domain. We leave that discussion to Section 3. Secondly, there is no restriction on the types which can be used as template parameter specializations of the ground ring. Any type used as a specialization of this ground ring template parameter should truly be a ring and not any other nonsense type. Recall, it should be hard to do the wrong thing.

Leveraging another template trick along with multiple inheritance, this can be solved with the so-called `Derived_from` class[2] which determines at compile-time if one class is the subclass of another. `Derived_from` is a template class with two parameters: one a potential subclass, and the other a superclass. This class defines a function converting the apparent subclass type to the superclass. If the conversion is valid via implicit up-casting, then the function is well-formed, otherwise, a compiler error occurs.

To make use of `Derived_from`, a class template inherits from `Derived_from`, passing its own template parameter to `Derived_from` as the potential subclass, along with a statically defined superclass type. This enforces that the template parameter be a subclass of that superclass. In our implementation, shown in Listing 2, polynomial classes enforce that their ground ring should be a `BPASRing`, our abstract class for rings (recall the declaration of `BPASRing` from Listing 1).

```
// If T is not a subclass of Base, a compiler error occurs
template <class T, class Base> class Derived_from {
    static void constraints(T* p) { Base* pb = p; }
    Derived_from() { void(*p)(T*) = constraints; }
};

template <class Ring, class Derived>
class BPASPolynomial : BPASRing<Derived>, Derived_from<Ring, BPASRing<Ring>>;
```

Listing 2: An implementation of a polynomial interface using CRTP and `Derived_from`.

All of these functionalities together create an algebraic hierarchy as a class hierarchy while maintaining strict type safety. Yet, our scheme remains flexible enough to support implicit conversions, such as natural ring embeddings, and generic enough to allow, for example, polynomials over user-defined classes, as long as those classes inherit from `BPASRing`. What remains now is to address the issue of polynomial rings sometimes forming different algebraic types depending on their particular ground ring.

3 "Dynamic" Type Creation, Conditional Export

In object-oriented design, the combination of types to create another type is known as composition. In this section, let us consider univariate polynomial rings; one can always work recursively for multivariate polynomials. Viewing a polynomial ring as a ring extension of its ground ring, polynomials can be seen as the composition of some finite number of elements of that ground ring. Moreover, we know that the properties of a polynomial ring depend on the properties of the ground ring. For example, the ring of univariate polynomials over a field is a Euclidean domain while the ring of polynomials over a ring is itself only a ring. Recall from the previous section that our implementation of polynomials are templated by their ground ring. Our goal then is to capture the idea that the position of a polynomial ring in the algebraic class hierarchy changes depending on the particular specialization of this template parameter.

More generally, we would like that the type resulting from the composition of another type depends on the type being composed. Hence, a sort-of "dynamic"

[2]`Derived_from` is a long-known trick, but is now adopted into the C++20 standard.

type creation. This is not truly dynamic, since it is a compile-time operation, but it nonetheless feels dynamic since it is an automatic process by the compiler via template instantiation. In fact, having this occur at compile-time is actually a benefit where errors can be determined preemptively. One can also view this mechanism as a way of controlling the methods which the newly created type exports. That is, conditionally exposing methods (or other attributes) in its interface depending on the particular template parameter specialization. This technique relies on compile-time introspection and SFINAE.

3.1 SFINAE and Compile-Time Introspection

Substitution Failure Is Not An Error (SFINAE), coined by Vandevoorde in [17], refers to a fundamental part of C++ templating. The invalid substitution of a type as a template parameter is itself not an error. Such a principle is required for templates to be practical. Where two or more template specializations exist, it is not required that the substitution of the template parameter fit all of the specializations, but only one. This principle, combined with compile-time function overload resolution, provides template metaprogramming its power. In particular, *compile-time introspection* is possible: using templates, truth values about a type can be determined and then made use of within the program.

Consider the typical example, adapted from [17, Section 8.3], shown in Listing 3. `type_has_X` determines if a type has a member `X` by checking the size of the return type of a function. By function overload resolution, if `T` has a member `X` the `test<T>` function chosen will be the first, whose return type has size 1. Otherwise the second function is chosen with return type of size (at least) 2.

```
template<typename T> char test(typename T::X const*);
template<typename T> int  test(...);
#define type_has_X(T) (sizeof(test<T>(NULL)) == 1);
```

Listing 3: A simple compile-time introspection to determine if type T has member X.

This idea can be generalized to many introspective metaprogramming techniques. For example, `is_base_of`, a standard feature in C++11, is much like `Derived_from`. However, instead of creating a compiler error, `is_base_of` determines a Boolean value representing if one type is derived from another.

Using introspection, one may think that `enable_if`, another standard C++11 template construct, is sufficient. The `enable_if` struct template conditionally compiles and exposes a function template based on the value of a Boolean known at compile-time. This Boolean value can of course be determined by introspection. Unfortunately, function templates cannot be virtual, thus this solution cannot be used within a class hierarchy. Conditionally exposing methods in our algebraic class hierarchy requires a different solution.

3.2 Conditional Inheritance for Polynomials

Defining new types dependent on the value of another type, as well as conditionally exposing member functions, can both be fulfilled by *conditional inheritance*. Specifically, we implement a compile-time case discussion for inheritance based on introspective values. In the context of polynomials in our algebraic class hierarchy, that case discussion works as a cascade of type checks on the ground ring, say R, when forming the polynomial ring $R[x]$. For example: if R is a field, then $R[x]$ is a Euclidean domain; else if R is a GCD domain, so is $R[x]$; else if R is an integral domain, so is $R[x]$; else $R[x]$ is a ring. This case discussion can be extended to include as much granularity as needed.

To perform this case discussion, we use the C++11 metaprogramming feature `conditional`, which uses a Boolean value known at compile-time to choose between two types. This is much like the ternary conditional operator which uses a Boolean to choose between two statements. Using `is_base_of` to determine the Boolean, `conditional` chooses one of two types to use as a class's superclass.

As a simple example, consider Listing 4. The definition of `BPASPolynomial` tests if the `Ring` template parameter is a subclass of `BPASField`. If so, `conditional` chooses `BPASEuclideanDomain` as the the superclass of `BPASPolynomial`, otherwise `BPASRing` is chosen. Additionally, a concrete class `SparseUnivarPoly` is shown, still parameterized by a coefficient ring. In this concrete class, the interface of the class will adapt "dynamically" to a particular template specialization via the `conditional` in its superclass. Notice also that the template parameter of `SparseUnivarPoly` is enforced to be a subclass of `BPASRing` on specialization via the `Derived_from` of its superclass.

```
template <class Ring, class Derived>
class BPASPolynomial : conditional< is_base_of<Ring, BPASField<Ring>>::value,
                                    BPASEuclideanDomain<Derived>,
                                    BPASRing<Derived> >::type,
                       Derived_from< Ring, BPASRing<Ring> >;

template <class CoefRing>
class SparseUnivarPoly : BPASPolynomial<CoefRing,SparseUnivarPoly<CoefRing>>;
```

Listing 4: A simple use of `conditional` to choose between Euclidean domain or ring as the algebraic type of a polynomial based on its template parameter.

The presented code for `BPASPolynomial` in Listing 4 is rather simple, showing only a single type check. To implement a chain of type checks, the "else" branch of a `conditional` should simply be another `conditional`. To improve the readability of this case discussion, we avoid directly implementing nested if-else chains, and thus avoid using one `conditional` inside another. Instead, we create two symmetric class hierarchies, one representing the true algebraic class inclusions while the other is a "tester" hierarchy.

This tester hierarchy uses one `conditional` to determine if a property holds and, if so, chooses the corresponding class from the algebraic hierarchy as superclass. Otherwise, the next tester in the hierarchy is chosen as superclass to trigger the evaluation of the next `conditional`. Finally, all concrete polynomial classes inherit from `BPASPolynomial` to automatically determine their correct interface based on their ground ring. This structure is shown in Figure 1, with the algebraic hierarchy on the left, and the tester hierarchy on the right.

This technique of conditional inheritance is a powerful tool in any class template hierarchy. By understanding the properties of a type via introspection, it can automatically be incorporated into an existing class hierarchy either as itself or when used in composition to create a new type. For example, based on the specialization of a template parameter, the definition of a class template can be changed automat-

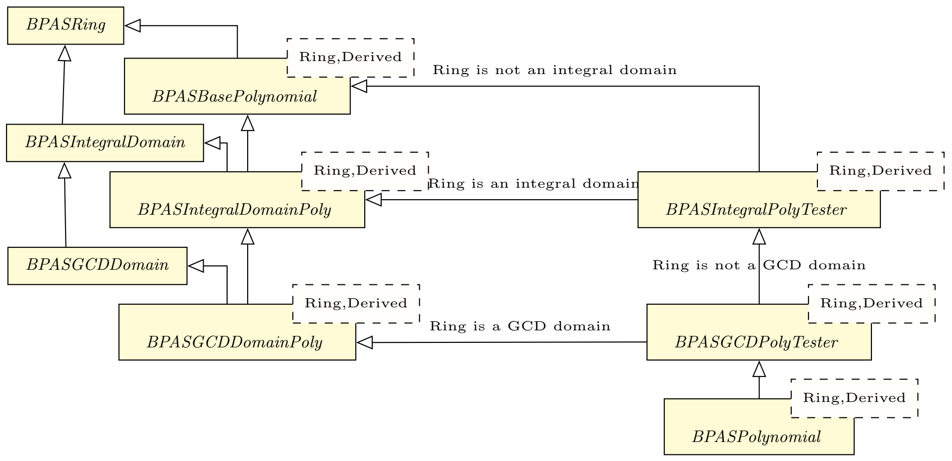

Figure 1: UML diagram for a subset of the polynomial abstract class hierarchy. Recall in UML that template parameters are shown in dashed boxes. Template parameters for non-polynomial classes are omitted for clarity. Note also that the multiple inheritance diamond problem is easily solved using virtual inheritance.

ically and dynamically. Not only does this enforce a proper class interface, but it allows the possibility of choosing between several different abstract implementations in order to best support the new type (i.e. the result of a composition).

4 Discussion and Related Work

For decades, computer algebra systems have worked towards type safety. Axiom [12] is a pioneering work on that front, but has grown out of popularity. Functional languages, like Scala and Haskell, have seen some progress in developing computer algebra systems thanks to type classes (see, e.g., [13] and references therein). These languages and their type classes provide a very suitable environment to define algebraic structures. However, while powerful, functional languages can be seen as an obscure and inaccessible programming paradigm compared to the mainstream imperative paradigm.

Considering other C/C++ computer algebra libraries, there are many examples with interesting mechanisms for handling algebraic structures. The SINGULAR library [9] perhaps has the most simple mechanism: a single class represents all rings, using a number of enum and Boolean variables to determine properties of instances at runtime. In CoCoALib [1] an abstract base class `RingBase` declares many functions returning Boolean values. Concrete subclasses define these functions to determine properties at runtime. While rings are subclasses of `RingBase`, elements of a ring are an entirely different class. Elements have pointers to the ring they belong, which are then compared at runtime to ensure compatibility in arithmetic between two elements. LinBox [15] also has separate classes for rings and their elements. There, ring properties are encoded as class templates where concrete rings use explicit template specialization to define properties.

Much like the previous cases, the MATHEMAGIX system requires instances (i.e. elements) of a ring to have a specific reference to a separate entity encoding the ring itself. Notably, MATHEMAGIX also includes a scheme to import and export C++ code to and from the MATHEMAGIX language [16]. This uses templates to allow, for example, a ring specified in the MATHEMAGIX language to be used as the coefficient ring for polynomials defined in C++.

In all of these cases there is some limiting factor. Most often, mathematical type safety is only a runtime property maintained by checking values. In some cases this is implemented by separating rings themselves from elements of a ring, a process counterintuitive to object-oriented design where one class should define the behaviour of all instances of that type.

On the contrary, our scheme does not rely on runtime checks. Instead, a function declaration itself restricts its arguments to be mathematically compatible at compile-time via the use of template parameters and the Curiously Recurring Template Pattern. By using an abstract class hierarchy many such function declarations are combined through consecutive inheritances to build up an interface incrementally. This closely follows the chain of class inclusions for algebraic types, where each type adds properties to the previous. The symmetry between the algebraic hierarchy and our class hierarchy hopes to make our interfaces natural and approachable to an end-user. This symmetry comes at the price of creating a deep class hierarchy, and thus strong coupling within the class hierarchy. Yet, this price is worth the symmetry and comprehensibility of the class hierarchy with the algebraic hierarchy.

In contrast with our class hierarchy solution to type safety, a different compile-time solution could be crafted through further use of type traits (see, e.g., [17, Ch. 15, 17]). Type traits are template metaprogramming constructs for type introspection and modification, some of which have already been seen, such as `is_base_of`, and `conditional`. Type traits are arguably more flexible, however, template metaprogramming is already rather difficult, and is essentially limited to C++. Class hierarchies, on the other hand, are present in every object-oriented language and should therefore be more accessible to end-users. The use of class hierarchies, in addition to encapsulating much of the template metaprogramming in our design, should provide better extensibility to end-users in general.

5 Conclusion and Future Work

In this work we have explored part of the implementation and design of the C++ interface of the BPAS library. Through the use of template metaprogramming we have devised a so-called algebraic class hierarchy which directly models the algebraic hierarchy while providing compile-time type safety. This hierarchy is type-safe both in the programming language sense and the mathematical sense.

Using inheritance throughout the algebraic abstract class hierarchy, the interface of algebraic types is constructed incrementally. Therefore, a concrete type's properties and interface is determined by its particular abstract superclass from this hierarchy. Through additional templating techniques, we can automatically infer, at compile-time, the correct superclass (and thus interface) of new types created by template parameter specialization (e.g. polynomials). The result is a consistent and enforced interface for all classes modelling algebraic types.

We are currently working to extend our algebraic class hierarchy to include multivariate power series, polynomials with power series coefficients, and polynomials in prime characteristic. This more capable hierarchy will be used within our library to implement a sophisticated solver for systems of polynomial equations, a prototype of which is already available in recent releases of BPAS. Finally, we hope to create a Python interface to the BPAS library (i.e. an extension module) to further improve the accessibility and ease of use of our library.

Acknowledgements

The authors would like to thank IBM Canada Ltd (CAS project 880) and NSERC of Canada (CRD grant CRDPJ500717-16, award CGSD3-535362-2019).

References

[1] J. Abbott and A. M. Bigatti. *CoCoALib: a C++ library for doing Computations in Commutative Algebra*. Available at `http://cocoa.dima.unige.it/cocoalib`.

[2] M. Asadi, A. Brandt, C. Chen, S. Covanov, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, L. Wang, N. Xie, and Y. Xie. *Basic Polynomial Algebra Subprograms (BPAS)*. `http://bpaslib.org`. 2020.

[3] M. Asadi, A. Brandt, R. H. C. Moir, and M. Moreno Maza. "Algorithms and Data Structures for Sparse Polynomial Arithmetic". In: *Mathematics* 7.5 (2019), p. 441.

[4] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. Xie. "On the Parallelization of Triangular Decomposition of Polynomial Systems". In: *CoRR* abs/1906.00039 (2019).

[5] L. Bernardin, P. Chin, P. DeMarco, K. O. Geddes, D. E. G. Hare, K. Heal, G. Labahn, J. P. May, J. McCarron, M. B. Monagan, D. Ohashi, and S. M. Vorkoetter. *Maple Programming Guide*. `www.maplesoft.com/documentation_center/maple2018/ProgrammingGuide.pdf`. 2018.

[6] W. Bosma, J. Cannon, and C. Playoust. "The Magma algebra system. I. The user language". In: *J. Symbolic Comput.* 24.3-4 (1997). Computational algebra and number theory (London, 1993), pp. 235–265.

[7] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. "The Basic Polynomial Algebra Subprograms". In: *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*. 2014, pp. 669–676.

[8] S. Covanov, D. Mohajerani, M. Moreno Maza, and L. Wang. "Big Prime Field FFT on Multi-core Processors". In: *2019 International Symposium on Symbolic and Algebraic Computation, Proceedings*. 2019, pp. 106–113.

[9] W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann. Singular *4-1-1 — A computer algebra system for polynomial computations*. `http://www.singular.uni-kl.de`. 2018.

[10] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik. "How do api documentation and static typing affect api usability?" In: *36th International Conference on Software Engineering, Proceedings*. ACM. 2014, pp. 632–642.

[11] W. Hart, F. Johansson, and S. Pancratz. *FLINT: Fast Library for Number Theory*. Version 2.5.2, `http://flintlib.org`. 2015.

[12] R. D. Jenks and R. S. Sutor. *Axiom, the scientific computation system*. 1992.

[13] R. Jolly. "Categories as Type Classes in the Scala Algebra System". In: *Computer Algebra in Scientific Computing - 15th International Workshop, Proceedings*. 2013, pp. 209–218.

[14] V. Shoup et al. *NTL: A library for doing number theory*. `www.shoup.net/ntl/`.

[15] The LinBox group. *LinBox*. v1.6.3. 2019. URL: `http://github.com/linbox-team/linbox`.

[16] J. van der Hoeven and G. Lecerf. "Interfacing mathemagix with C++". In: *Proceedings of the 2013 International Symposium on Symbolic and Algebraic Computation*. 2013, pp. 363–370.

[17] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[18] Wolfram Research, Inc. *Mathematica, Version 11.3*. Champaign, IL, 2018.