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

Basic Polynomial Algebra Subprograms

Alexander Brandt, Robert H. C. Moir, Marc Moreno Maza

Ontario Research Center for Computer Algebra
Department of Computer Science
University of Western Ontario, Canada

ICMS 2020

The BPAS Library

Basic Polynomial Algebra Subprograms (BPAS) [2] provides **high-performance polynomial algebra**.

→ High performance: core implementation in C considers data locality, cache complexity, and parallelism for modern multi-core architectures

→ Easy to use: **"Dynamic" Object-Oriented interface** in C++ is a light-weight wrapper of the underlying, optimized C code.

Notable highly-optimized operations include:

→ FFTs, parallel integer polynomial multiplication, modular polynomial arithmetic, parallel Taylor shift, real root isolation (ICMS 2014 [5])

→ Big prime field FFTs, arithmetic in $\mathbb{Z}/p\mathbb{Z}$ for large characteristic [6]

→ Sparse polynomial arithmetic, pseudo-division, normal form [4]

→ Polynomial system solving: parallel triangular decomposition via regular chains [3]

Outline

Motivation: Usability

BPAS is concerned with **accessibility**, **interoperability**, and **usability**.

→ Open-source and written in C/C++ provides the former two.

To achieve usability, we consider best practices for its interface.

1. Natural: a symmetric encoding of the algebraic hierarchy

   field ⊂ Euclidean domain ⊂ GCD domain ⊂ integral domain ⊂ ring

2. Easy to use: an object-oriented design with well-defined interfaces. A so-called **algebraic class hierarchy**: rings are classes and elements of a ring are objects

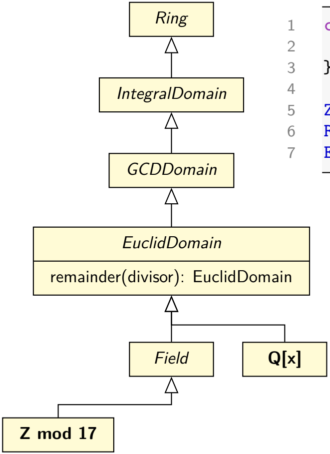
3. Encapsulation: hide complexity of low-level code; class interfaces

4. Extensible: adaptable to new (user-created) types, type composition

5. Type safe: compile-time type safety and mathematical type safety

Motivation: Type Safety

A naive implementation of the algebraic hierarchy as a class hierarchy creates mathematically unsafe operations via polymorphism.




```
class EuclidDomain {
    EuclidDomain remainder(EuclidDomain& divisor);
}

Zmod17 a;
RationalPoly b;
EuclidDomain r = a.remainder(b);
```

→ $\mathbb{Z}/17\mathbb{Z}$ and $\mathbb{Q}[x]$ are Euclidean domains

→ the code is valid via polymorphism

→ could compile, but then issues at runtime.

Existing Solutions

In other compiled libraries, mathematical type safety is only a runtime property maintained through runtime value checks.

→ In Singular's libpolys [7], all algebraic types are a single class. Instance variables (Booleans, enums) store properties of rings

→ In CoCoA [1] rings and elements of a ring are separate classes. Elements hold references to their "owning" ring which are compared at runtime and errors thrown if not identical.

→ In LinBox [8] rings and elements are again distinct, with references to abstract ring elements being downcasted for operations.

Our Goal: provide both compile-time mathematical type safety and a natural, extensible object-oriented hierarchy for the algebraic hierarchy

Outline

Templates in C++

A **class template** in C++ is a parameterized class definition

→ *template instantiation*: providing a particular value for a template parameter

→ **compile-time** code generation and *overload resolution* occurs

→ synonymous definitions for function templates

Template metaprogramming uses templates to control and modify a program's code or compilation

→ facilitates compile-time code generation

→ facilitates **compile-time code evaluation**

Compile-Time Introspection

Templates allow for *compile-time introspection*

→ compile-time evaluation of code to determine properties of a type

```
struct Foo {
    typedef int X;
};

template<typename T> char test(typename T::X const*);

template<typename T> int test(...);

#define type_has_X(T) (sizeof(test<T>(NULL)) == 1);

std::cout << "Foo has X: " << type_has_X(Foo);
std::cout << "int has X: " << type_has_X(int);
```

→ if `T` has type `X`, then compile-time overload resolution chooses first definition with return type `char` (size == 1)

→ else, second definition is chosen with return type `int` (size >= 2)

Outline

Algebraic Class Hierarchy

The algebraic hierarchy as a class hierarchy with mathematical type safety

Solution: an **abstract class template hierarchy**.

- → abstract classes: well-defined interfaces, default behaviour
- → inheritance incrementally extends/builds interface
- → template parameter modifies interface to restrict method parameters

```cpp
template <class Derived>
class Ring {...};

template <class Derived>
class IntegralDomain : Ring<Derived> {...};

template <class Derived>
class GCDDomain : IntegralDomain<Derived> {...};

template <class Derived>
class EuclidDomain : GCDDomain<Derived> {
    Derived remainder(Derived& divisor);
}
```

Algebraic Class Hierarchy: Static Polymorphism

Static polymorphism via *Curiously Recurring Template Pattern*: concrete class is used as template parameter of super class.

→ function resolution occurs at compile-time

→ method declaration restricts params to be compile-time compatible

```cpp
template <class Derived>
class EuclidDomain : GCDDomain<Derived> {
    Derived remainder(const Derived& divisor);
};

class Integer : EuclidDomain<Integer> {...}; //CRTP
//Integer remainder(const Integer& divisor);

class RationalPoly : EuclidDomain<RatonalPoly> {...}; //CRTP
//RationalPoly remainder(const RationalPoly& divisor);

Integer x;  RationalPoly p;

//compiler error: EuclidDomain<RationalPoly>::remainder
//                takes RationalPoly as parameter
RationalPoly r = p.remainder(x);
```

Algebraic Class Hierarchy: Adding Flexibility

Disjoint class hierarchies is too restrictive. Allow *implicit conversion* by defining constructors, e.g. for natural ring embeddings $\mathbb{Z} \hookrightarrow \mathbb{Q} \hookrightarrow \mathbb{Q}[x]$

```
class Integer : EuclidDomain<Integer> {};

class RationalPoly : EuclidDomain<RatonalPoly> {
    RationalPoly(Integer x) {...};
};

Integer x;
RationalPoly p;

//no error: implicit conversion, Integer to RationalPoly
RationalPoly r = p.remainder(x);
```

→ Explicitly defining constructors gives permission for compatibility between types at compile-time

→ *cf.* working in a restrictive manner: allow everything at compile-time and catch incompatibility at runtime

Outline

Algebraic Class Hierarchy with Polynomials

Extend abstract class template hierarchy to include polynomials

→ parameterize polynomial abstract classes by coefficient ring

```
template <class Derived>
class Ring {...};

template <class CoefRing, class Derived>
class Poly : Ring<Derived> {...};

class RationalPoly : Poly<RationalNumber, RationalPoly> {...};
```

Problem: What if `CoefRing` is not actually a ring?

→ e.g. `Poly<std::string>` or `Poly::<Apple>`

Problem: polynomial rings form different algebraic types depending on the ground ring

→ e.g. $\mathbb{Q}[x]$ is a Euclidean domain, $\mathbb{Z}[x]$ is an integral domain

Constraining the Ground Ring

At compile-time ensure that a polynomial's coefficient ring is an actual ring with template metaprogramming.

`Derived_from<T, Base>`: statically determines if `T` is a subclass of `Base`, creating a compiler-error if not

→ inheriting from `Derived_from` forces evaluation at compile-time during template instantiation

→ Coefficient ring must be a subclass of `Ring`

→ `Poly` can assume `CoefRing` has a certain interface at minimum

```
template <class T, class Base>
class Derived_from {...};

template <class CoefRing, class Derived>
class Poly : Ring<Derived>,
             Derived_from<CoefRing, Ring<CoefRing>> {...};
```

Adapting to Different Coefficient Rings (1/2)

Determine type of coefficient ring using *compile-time introspection*

- → **Conditional inheritance** then determines correct algebraic type and interface for polynomials over that ring
- → "Dynamic" type creation via introspection, template instantiation

`is_base_of<T, Base>::value`

- → compile-time Boolean value determines if `T` is a subclass of `Base`

`conditional<Bool, T1, T2>::value`

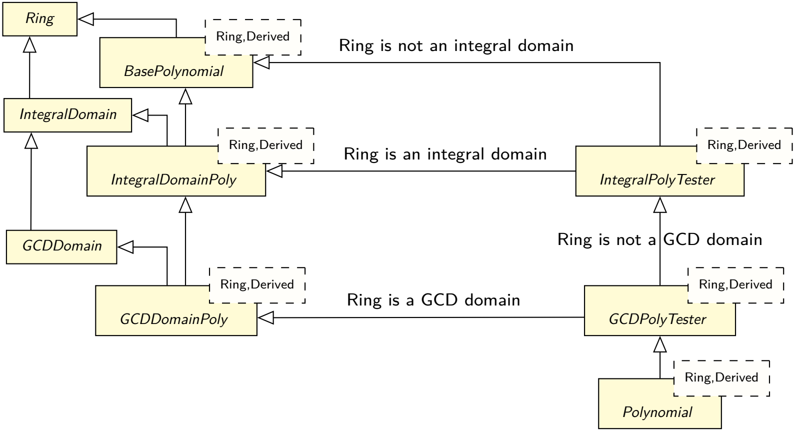
- → A compile-time tertiary conditional operator for choosing types
- → `Bool ? T1 : T2`

```
template <class CRing, class Derived>
class Poly : conditional< is_base_of<CRing, Field<CRing>>::value,
                          EuclidDomain<Derived>,
                          Ring<Derived>
                        >::value {...};
```

Adapting to Different Coefficient Rings (2/2)

A chain of `conditional`'s create a case-discussion at compile-time

- → *Tester* hierarchy separates introspection from actual interface
- → Concrete classes inherit from *Polynomial* to automatically determine their type and interface



Conclusions and Future Work

Algebraic Class Hierarchy as an abstract class template hierarchy

→ Direct object-oriented encoding of algebraic types for strict interfaces

→ Compile-time type safety, implicit conversion allows compatibility

→ Properties of rings (classes) can be exploited with introspection

→ Adaptive polynomial class hierarchy through conditional inheritance

↳ More generally, conditionally exposes/adds methods to an interface

In future:

→ The hierarchy will be extended to include power series, polynomials over prime fields

→ A Python interface will be added on top of the C++ interface

Thank You!

I look forward to your questions:

→ during the live Q/A session, or

→ via email, Alex Brandt <abrandt5@uwo.ca>

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, M. Kazemi, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, L. Wang, N. Xie, and Y. Xie. *Basic Polynomial Algebra Subprograms (BPAS)*. http://www.bpaslib.org. 2020.

[3] M. Asadi, A. Brandt, R. H. C. Moir, M. Moreno Maza, and Y. X. "On the Parallelization of Triangular Decomposition of Polynomial Systems". In: *ISSAC 2020, Proceedings*. (to appear). 2020.

[4] 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.

[5] C. Chen, S. Covanov, F. Mansouri, M. Moreno Maza, N. Xie, and Y. Xie. "The Basic Polynomial Algebra Subprograms". In: *ICMS 2014, Proceedings*. 2014, pp. 669–676.

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

[7] 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.

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