Evaluation of a Java Computer Algebra System

Heinz Kredel

IT-Center, University of Mannheim, 68131 Mannheim, Germany
kredel@rz.uni-mannheim.de

Abstract. This paper evaluates the suitability of Java as an implementation language for the foundations of a computer algebra library. The design of basic arithmetic and multivariate polynomial interfaces and classes have been presented in [1]. The library is type-safe due to its design with Java's generic type parameters and thread-safe using Java's concurrent programming facilities. We evaluate some key points of our library and differences to other computer algebra systems.

1 Introduction

We have presented an object oriented design of a Java Computer Algebra System (called JAS in the following) as type safe and thread safe approach to computer algebra in [1–3]. JAS provides a well designed library for algebraic computations implemented with the aid of Java's generic types. The library can be used as any other Java software package or it can be used interactively or interpreted through an jython (Java Python) front end. The focus of JAS is at the moment on commutative and solvable polynomials, Groebner bases and applications. By the use of Java as implementation language, JAS is 64-bit and multi-core cpu ready. JAS is being developed since 2000 (see the weblog in [3]).

This work is interesting for computer science and mathematics, since it explores the Java [4] type system for expressiveness and eventual short comings. Moreover it employs many Java packages, and stresses their design and performance in the context of computer algebra, in competition with systems implemented in other programming languages.

JAS contains interfaces and classes for basic arithmetic of, e.g. integers and rational numbers and multivariate polynomials with such coefficients. Additional packages in JAS are:

- The package `edu.jas.ufd` contains classes for unique factorization domains. Like the interface `GreatestCommonDivisor`, an abstract class providing commonly useful methods and classes with implementations for polynomial remainder sequences and modular algorithms.
- The package `edu.jas.ring` contains classes for polynomial and solvable polynomial reduction, Groebner bases and ideal arithmetic as well as thread parallel and distributed versions of Buchbergers algorithm like `ReductionSeq`, `GroebnerBaseParallel` and `GroebnerBaseDistributed`.
- The package `edu.jas.module` contains classes for module Groebner bases, syzygies for polynomials and solvable polynomials like `ModGroebnerBase` or `SolvableSyzygy`.
- Finally, the package `edu.jas.application` contains applications of Groebner bases, such as ideal intersections and ideal quotients in the classes `Ideal` or `SolvableIdeal`.

The emphasis of this paper is the evaluation of the JAS library design with respect to the points: interfaces as types, generics and inheritance, dependent types, method semantics, recursive types, design patterns, code reuse, performance, applications, parallelization, libraries, and the Java environment.

1.1 Related Work

In this section we briefly give some pointers to related work, for details see [1]. For an overview on other computer algebra systems see [5]. Typed computer algebra systems with own programming languages are described e.g. in [6], [7] and [8]. Computer algebra systems implemented in other programming languages and libraries are: in C/C++ [9–11], in Modula-2 [12], in Oberon [13] and in FORTRAN [14]. Java computer algebra implementations have been discussed in [15], [16], [17], [18], [19] and [20]. Newer approaches are discussed in [21], [22] and [23]. The expression of mathematical requirements for generic algorithms in programming language constructs have been discussed in [24] and [25].

More related work, together with an evaluation of the design, is discussed in section 3. Due to limited space we have not discussed the related mathematical work on solvable polynomials, Groebner base and greatest common divisor algorithms, see e.g. [26, 27] for some introduction. This paper contains an expanded, revised and corrected part of [2].

1.2 Outline

In the next section 2, we give some examples on using the JAS library and give an overview of the JAS type system for polynomials. Section 3 evaluates the presented design and compares JAS to other computer algebra systems. In particular it discusses interfaces as types, generics and inheritance, dependent types, method semantics, recursive types, design patterns, code reuse, performance, applications, parallelization, libraries, and the Java development environment. Finally section 4 draws some conclusions.

```
T[0] = 1
T[1] = x
T[2] = 2 x^2 - 1
T[3] = 4 x^3 - 3 x
T[4] = 8 x^4 - 8 x^2 + 1
T[5] = 16 x^5 - 20 x^3 + 5 x
T[6] = 32 x^6 - 48 x^4 + 18 x^2 - 1
T[7] = 64 x^7 - 112 x^5 + 56 x^3 - 7 x
T[8] = 128 x^8 - 256 x^6 + 160 x^4 - 32 x^2 + 1
T[9] = 256 x^9 - 576 x^7 + 432 x^5 - 120 x^3 + 9 x
```

Fig. 1. Chebychev polynomials

2 Introduction to JAS

In this section we show some examples for the usage of the JAS library, and then discuss the general layout of the polynomial types. Some parts of this section are similar to the JAS introduction in [3].

2.1 Using the JAS Library

To give a first idea about the usage of the library, we show the computation of Chebychev polynomials. They are defined by the recursion: $T[0] = 1$, $T[1] = x$, $T[n] = 2x \times T[n-1] - T[n-2] \in \mathbb{Z}[x]$. The first ten Chebychev polynomials are shown in figure 1.

```
 1.  int m = 10;
 2.  BigInteger fac = new BigInteger();
 3.  String[] var   = new String[]{ "x" };
 4.  GenPolynomialRing<BigInteger> ring
 5.                 = new GenPolynomialRing<BigInteger>(fac,1,var);
 6.  List<GenPolynomial<BigInteger>> T
 7.                 = new ArrayList<GenPolynomial<BigInteger>>(m);
 8.  GenPolynomial<BigInteger> t, one, x, x2;
 9.  one = ring.getONE();
10.  x   = ring.univariate(0); // polynomial in variable 0
11.  x2  = ring.parse("2 x");
12.  T.add( one );    // T[0]
13.  T.add( x );      // T[1]
14.  for ( int n = 2; n < m; n++ ) {
15.      t = x2.multiply( T.get(n-1) ).subtract( T.get(n-2) );
16.      T.add( t );   // T[n]
17.  }
18.  for ( int n = 0; n < m; n++ ) {
19.      System.out.println("T["+n+"] = " + T.get(n) );
20.  }
```

Fig. 2. Computing Chebychev polynomials

The polynomials have been computed with the Java program in figure 2. In lines 4 and 5 we construct a polynomial factory `ring` over the integers, in one variable `"x"`. This factory object itself needs at least a factory for the creation of coefficients and the number of variables. Additionally the term order and names for the variables can be specified. With this information the polynomial ring factory can be created by `new GenPolynomialRing <BigInteger> (fac,1,var)`, where `fac` is the coefficient factory, `1` is the number of variables, and `var` is an `String` array of names. In lines 8 to 11 the polynomials for the recursion base, `one` and `x` are created. Both are generated from the polynomial factory with method `ring.getONE()` and `ring.univariate(0)`, respectively. The polynomial $2x$ is, e.g. produced by the method `ring.parse("2 x")`. The string argument of method `parse()` can be the TEX-representation of the polynomial, except that no subscripts may appear. Note, `x2` could also be created

from the coefficient factory by using `x.multiply( fac.fromInteger(2) )` or, directly by `x.multiply( new BigInteger(2) )`.

In lines 6 and 7 a list `T` is defined and created to store the computed polynomials. Then, in the for-loop, the polynomials $T[n]$ are computed using the definition, and stored in the list `T` for further use. In the last for-loop, the polynomials are printed, producing the output shown in figure 1. The string representation of the polynomial object can be created, as expected, by `toString()`.

To use other coefficient rings, one simply changes the generic type parameter, say, from `BigInteger` to `BigComplex` and adjusts the coefficient factory. The factory would then be created as `c = new BigComplex()`, followed by `new GenPolynomialRing<BigComplex>(c,1,var)`. This small example shows that this library can easily be used, just as any other Java package or library.

2.2 JAS Class Overview

The central interface is `RingElem` (see figure 3, left part) which extends `AbelianGroupElem` with the additive methods and `MonoidElem` with the multiplicative methods. Both extend `Element` with methods needed by all types. `RingElem` is itself extended by `GcdRingElem`, which includes greatest common divisor methods and `StarRingElem`, containing methods related to (complex) conjugation.

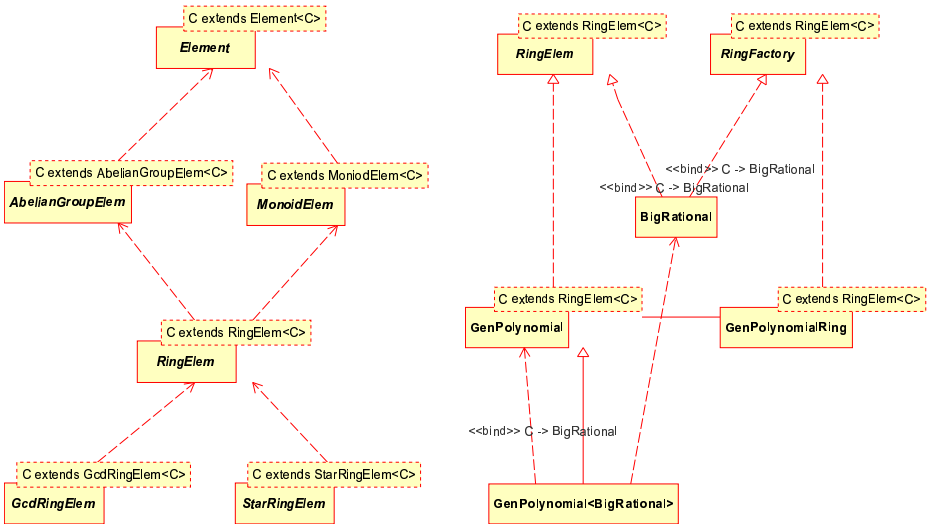

Fig. 3. Overview of some algebraic types and of generic polynomials

The interface `RingElem` defines a recursive type which defines the functionality of the polynomial coefficients and is also implemented by the polynomials. So polynomials can be taken as coefficients for other polynomials, thus defining a recursive polynomial ring structure. We separate the creational aspects of ring elements into ring factories with sufficient context information. The minimal factory functionality is defined by the interface `RingFactory` (see figure 3, right part). Constructors for polynomial rings will then require factories for the coefficients so that the construction of polynomials poses no problem.

The `RingElem` interface (with type parameter `C`), defines the commonly used methods required for ring arithmetic, such as `C sum(C S)`, `C subtract(C S)`, `C abs()`, `C negate()`, `C multiply(C s)`, `C divide(C s)`, `C remainder(C s)`, and `C inverse()`. In addition to the arithmetic methods defined by `RingElem`, there are testing methods such as `boolean isZERO()`, `isONE()`, `isUnit()` and `int signum()`. The first three test if the element is 0, 1 or a unit in the respective ring. The `signum()` method computes the sign of the element (in case of an ordered ring). The methods `equals(Object b)`, `int hashCode()` and `int compareTo(C b)` are required by Java's object machinery. The last method `C clone()` can be used to obtain a copy of the actual element.

The `RingFactory` interface defines the methods `C getZERO()`, `C getONE()`, which create 0 and 1 of the ring, respectively. The creation of the 1 is most difficult, since for a polynomial it implies the creation of the 1 from the coefficient ring, i.e. we need a factory for coefficients at this point. There are methods to embed a natural number into the ring and create the corresponding ring element, e.g. `C fromInteger(long a)`. Other important methods are `C random (int n)`, `C copy(C c)`, `C parse (String s)`, and `C parse (Reader r)`. The `random(int n)` method creates a random element of the respective ring. The two methods `parse(String s)` and `parse(Reader r)` create ring elements from some external representations. The methods `boolean isField()`, `isCommutative()` or `isAssociative()` query specific properties of the ring.

Generic polynomials are implemented in class `GenPolynomial`, which has a type parameter `C` that extends `RingElem<C>` for the coefficient type (see figure 3, right part). All operations on coefficients, that are required in polynomial arithmetic and manipulation are guaranteed to exist by the `RingElem` interface. The constructors of the polynomials always require a matching polynomial factory. The generic polynomial factory is implemented in the class `GenPolynomialRing`, again with type parameter `C extends RingElem<C>` (not `RingFactory`). The polynomial factory implements the interface `RingFactory<C extends RingElem<C>>` so that it can also be used as coefficient factory. The constructors for `GenPolynomialRing` require at least the parameters for a coefficient factory and the number of variables of the polynomial ring.

The design of the other types and classes together with aspects of implementation are discussed in detail in [1].

3 Evaluation

In this section we discuss, without striving for completeness, some key points of our library and differences to other systems. Due to space restrictions, we assume some knowledge of [1] in the following, see also [3] and the related work in the introduction.

3.1 Interfaces as types

In [28, 29] the authors argue, and give counter examples, that a type system based only on (multiple) inheritance, is not sufficient to implement mathematical libraries, in particular, it is not sufficient to implement computer algebra libraries. As a solution they advocate interfaces, called signatures in their paper, as we find them now in Java. With the aid of interfaces it is possible to define

a abstract type system separate of any implementation types defined by class hierarchies. This approach was partly anticipated in the Axiom system [6] with so called *categories* and *domains*. A *category* is a kind of interface, but with the possibility to give implementations for certain methods, like an Java abstract class. A *domain* in Axiom is similar to a Java class. In [30] the necessary flexibility for the type system was achieved by a decoupling of classes from so called *views* (interfaces). In defining *views*, one could however, give arbitrary mappings for the *view* method names to the implementing class method names. Java allows only exact matching names, or one has to resort to some facade pattern to map names during runtime. The definition of the type hierarchy from `Element` to `RingElem` is perfectly suited to abstract from the common characteristics of coefficients and polynomials to make them exchangeable and inter-operable.

In section 2.4 (*solvable polynomials*) in [1] a problem appeared with the type erasure the compiler does for generic types to generate the raw implementation classes. Further investigation revealed, that this is not a problem of type erasure, but a feature of any generic object oriented programming language. As, in general, a sub-class can not be allowed to implement a generic interface with a type parameter of the sub-class. Since this would require the compiler to check that no method of the super class, which is not overwritten in the sub-class, uses a super class constructor. This is not feasible to check for the compiler and impossible for separately compiled class libraries. In our case of the `GenPolynomial` super class we assured by using factory methods of the sub-class `GenSolvable-Polynomial` that any super class method returns objects of the sub-class. I.e. we changed the semantics of the super class methods to return sub-class objects but a compiler can not suss this. This implies that our proposal to solve this, in [2], is wrong.

3.2 Generics and inheritance

The first version of the JAS library was implemented without generic types [31]. One obvious reason was, that generics where only introduced to the Java language in JDK 1.5. But it was well known from papers, such as [32], that generics are not necessarily required, when the programming language has, or allows the construction of a well-designed type hierarchy. In our previous implementation (up to 2005) we employed a interface `Coefficient`, which was implemented by coefficient classes and used in the `Polynomial` interface. `Polynomial` also extends `Coefficient` and so, polynomials could be used as coefficients of other polynomials. The `Coefficient` interface has now become the `RingElem` interface. However, with such a non-generic approach one eventually looses some type safety, i.e. one could inadvertently multiply a polynomial with `BigInteger` coefficients by a polynomial with, say `BigRational` coefficients, and the compiler could not determine a problem. To prevent this, we had incorporated the name of the coefficient type in the name of the polynomial type, e.g. `RatPolynomial` or `IntPolynomial` in that release. A second reason for this first design was non existent coefficient factories, which could construct coefficients, say for the constant polynomial 1. Although the coefficient specific polynomials, e.g. `RatPolynomial`, have been extended from an abstract polynomial base class, e.g. `MapPolynomial`, it lead to much duplication of code. With the current generic type parameter approach we have removed all duplicate code for polynomial implementations.

Moreover the type of the polynomial is clearly visible from the denotation, e.g. `GenPolynomial<BigInteger>`, of polynomial variables.

3.3 Dependent types

The problem of dependent types is that we cannot distinguish polynomials in, say 3 variables from polynomials in, say 5 variables from their type. This carves a hole in the type safety of our library. I.e. the polynomial factories `GenPolynomialRing <BigInteger> (c, 3)` and `GenPolynomialRing <BigInteger> (c, 5)` could produce polynomials with the same type `GenPolynomial <BigInteger>`, but will most likely produce a run-time error, when, say, they are added together. Of course, the method `equals()` of the factory will correctly determine, that the rings are not equal, but the compiler is not able to infer this information and we are not able to denote hints.

This problem also occurs in the class `ModInteger` and `ModIntegerRing`. The type depends on the value of the modulus chosen. I.e. `ModIntegerRing(5)` and `ModIntegerRing(7)` are incompatible, but are denoted by the same type. Although the implementation of arithmetic methods of `ModInteger` will always choose the modulus of the first operand and therefore there will not be a run-time error, but this can lead to wrong results in applications.

The SmallTalk system [30] could use a elegant solution for this problem. Since types are first class objects, they can be manipulated as any other object in the language. E.g. one could define the following (in Java like syntax)

```
class Mod7 = ModIntegerRing(7);
Mod7 x = new Mod7(1);
```

Now `Mod7` is a type, which can be used to define and create new objects.

A minor problem of the same kind occurs with the term order defined in the polynomial factory, see [1]. It too, could be incompatible at run-time and this fact it is not expressed in the type. The actual implementation ignores this problem and arithmetic methods will produce a correct result polynomial, with a term order chosen from one of its input polynomials. However applications could eventually be confused by this behavior, e.g. Groebner base calculations.

Other computer algebra systems, e.g. [6], treat the polynomial dependent type case with some coercion facility. I.e. in most cases it is clear how to coerce a polynomial in 3 variables to a polynomial in 5 variables by adding variables with exponent zero or to coerce both to polynomials in 8 variables if variable names are disjoint.

A type correct solution to the dependent type problem in Java would be, to introduce an new type for every expected variable number, e.g. `Var1`, `Var2`, `Var3`, and to use this as additional type parameter for polynomials

```
GenPolynomialRing<BigRational,Var5>.
```

The types `Var*` could be implemented by interfaces or more suitably by extension of an abstract base class defining an abstract method `numberOfVariables` which could be used to query the number of variables at runtime. However, such a solution is impractical, since the number of variables of polynomials in applications is often determined at run-time and not during compile time.

How dependent types can correctly be handled in a programming language by the compiler, is discussed in [33].

3.4 Method semantics

The interface `RingElem` defines several methods which cannot be implemented semantically correct in all classes. E.g.

- the method `signum()` makes no sense in unordered rings,
- the methods `divide()` and `remainder()` are not defined, if the divisor is zero or only of limited value for multivariate polynomials,
- the method `inverse()` may fail, if the element is not invertible, e.g. for `r = new ModIntegerRing(6)`, `a = new ModInteger(r,3)`, `a.inverse()` fails, since 3 is not invertible in Z_6.

More examples for other systems can be found in [34]. We have adopted the policy to allow any meaningful reaction in these cases. E.g. the method `signum()` in `BigComplex` returns 0 if the number is equal to 0 and a non zero value otherwise. The case of division by zero is in Java usually handled by throwing a run-time exception, and so do we. This is meaningful, since such a case is mostly an input error, which should have been handled by the calling programs.

For `inverse()`, the situation is slightly different. If the element is zero it is an error and a run-time exception can be thrown. But in the context of dependent types there are elements, which are not zero, but can nevertheless not be inverted. As in the above example 3 is not zero, but is not invertible in Z_6. Also matrices can be non-zero but are eventually not invertible. In Axiom [6] such cases are handled by returning a special constant `"failed"`, with obvious problems arising for the type system. In Java we have the mechanism of checked exceptions. So for `inverse()`, it should be considered to add a `throws` clause in the definition, to make the user aware of some potential problem. We will explore this concept in future refactorings of the library.

In JAS there are testing methods to determine such cases. E.g. `isZERO()` or `isUnit()` to check if an element is invertible. For `isUnit()` the computing time can be as high as the computing time for `inverse()`, which doubles the computing time at this point and may not always be practical. In the factories there are methods to check further conditions. E.g. `isField()`, to test if the ring is a field and therefore if all non-zero elements are invertible.

There are proposals in [24, 25] to formalize the semantic requirements for methods and types, so that the compiler can check them during compilation. Also Axiom [6] has some capabilities to specify and check method constraints. In Java we have a rudimentary possibility in the form of assertions to check constrains at run-time. We have not further explored this subject in JAS yet.

3.5 Recursive types

We have exercised some care in the definition of our interfaces to assure, that we can define recursive polynomials. First, the interface `RingElem` is defined as

```
RingElem<C extends RingElem<C>>.
```

So the type parameter `C` can (and must) itself be a subtype of `RingElem`. For polynomials we can define a polynomial with polynomials as coefficients

```
GenPolynomial< GenPolynomial<BigRational> >
```

In some applications presented in [1], e.g. the Groebner base algorithms, we make no use of this feature. However, there are many algebraic algorithms which are only meaningful in a recursive setting. E.g. greatest common divisors or factorization of multivariate polynomials. Although a study of this will be covered by a future publication, one observation is, that our type system will unfortunately lead to code duplication. The code for `baseGcd()` and `recursiveGcd()` is practically the same, but because of the recursive type system, the methods must have different parameter types. Further, by the type erasure problem mentioned above, they must also have different names.

We have successfully implemented a greatest common divisor package for multivariate polynomials, using these recursive type features. There is a clean interface `GreatestCommonDivisor` with only the most useful methods. These are `gcd()`, `lcm()`, `squarefreePart()`, `squarefreeFactors()`, `content()`, `primitivePart()`, and `resultant()`. The generic algorithms then work for all implemented field coefficients.

The abstract class `GreatestCommonDivisorAbstract` implements the full set of methods, specified in the interface. Only two methods must be implemented for the different gcd algorithms. The abstract class should eventually, be refactored to provide an abstract class for PRS algorithms and an abstract class for the modular algorithms. Details on the problems and solutions of this package will be covered by a future publication.

3.6 Factory pattern

The usage of the factory pattern to create objects with complex parametrization requirements is a standard technique in object oriented programming. Surprisingly, it has already been used in the SmallTalk implementation presented in [30]. Recently this approach was advocated again in [16, 17]. But, otherwise we see this pattern seldom in computer algebra systems. The mainly used way to create polynomials or matrices is via constructors or by coercions from general algebraic expressions [6].

There is a nice application of the factory pattern in the `ufd` package. The factory `GCDFactory` can be used to select one from the many greatest common divisor implementations. This allows non-experts of computer algebra to choose the right algorithm for their problem.

```
GreatestCommonDivisor<C> engine
                        = GCDFactory.<C>getImplementation( fac );
c = engine.gcd(a,b);
```

The (static) method `getImplementation()` constructs a suitable gcd implementation for the given type. The selection of the `getImplementation()` method takes place at compile time. I.e. depending on the desired actual type for `C`, different methods are selected. The coefficient factory parameter `fac` is used at run-time to check e.g. if the coefficients are from a field, to further optimize the selection. For the special cases of `BigInteger` and `ModInteger` coefficients, the modular algorithms can be employed.

This factory approach contrasts the approach taken in [24] and [25] to provide programming language constructs to specify the requirements for the employment of certain implementations. These constructs would then direct the

selection of the right algorithm. However, the authors conclude, that not all properties can be captured at compile-time and some tests have to be post phoned to run-time.

3.7 Code reuse

With the help of generic programming we could drastically reduce the code of the earlier MAS [12] (and of the SAC-2 [14]) libraries. MAS has three major polynomial implementations, called distributive and recursive representation, and univariate dense representation. For each representation there are three or more implementations. One 'class' for integer coefficients, one for rational number coefficients and one for modular integer coefficients. In JAS there is only one polynomial class, which works for all implemented coefficients.

In MAS, a so called *arbitrary domain polynomial* implementation exists. Here, the coefficients consist of a *domain descriptor* and a *domain value*. With the domain descriptor it was possible to select at run-time the corresponding implementation for the domain values and provide further context information for the called 'methods'. 13 coefficient domains have been implemented. Besides the lack of type safety, the introduction of a new coefficient implementation required the update of dispatching tables for all methods. The run-time selection of the implementation added a performance penalty (of about 20%), see [12] and the references there. With Java we have no performance loss for the generic coefficients and no need for recoding if new coefficients are introduced in the future.

A particular nice example for code reuse is the computation of e-Groebner bases, see e.g. [26]. They are d-Groebner bases but with e-reduction instead of d-reduction. This can be expressed by subclassing with a different constructor.

```
public class EGroebnerBaseSeq<C extends RingElem<C>>
             extends DGroebnerBaseSeq<C> {
      public EGroebnerBaseSeq(EReductionSeq<C> red) { ... }    }
```

3.8 Performance

The performance of generic programming implementations in C++, Java and C# compared to special hand-coded programs is discussed in [35]. As usual the authors conclude, that hand-coded programs are faster, but diminish the software engineering benefits. A conclusion, we do not share. For our polynomial implementation (see [1]), performance is mainly influenced by

1. the performance of coefficient arithmetic,
2. the sorted map implementation and
3. the exponent vector arithmetic.

Coefficient arithmetic of polynomials is based on the Java `BigInteger` class. `BigInteger` was initially implemented by a native C library, but since JDK 1.3 it is implemented in pure Java [36]. The new implementation now has better performance than the C library. Sorted map implementation is from the Java collections package, which uses known efficient algorithms for this purpose, and it is comparable to other libraries, such as the C++ STL. However, we are not aware of general performance comparisons of the collection frameworks.

The exponent vector implementation is based on Java arrays of `long`s. It could be faster by using arrays of `int`s or `short`s as most computations seldom require polynomials of degrees larger than 2^{16}. This would reduce the memory footprint of a polynomial and so improve cache performance. If Java would allow elementary types as type parameters, it would be possible to make the `ExpVector` class generic, e.g. `ExpVector<long>` or `ExpVector<int>`. However, using objects like `Long` or `Integer` as exponents, would imply auto-boxing and introduce too much performance penalties. To make the library useful for a wide range of applications we decided to stay with the implementation using `long`s.

Table 1. JAS polynomial multiplication benchmark

Computing times in seconds on AMD 1.6 GHz CPU.
Options are: coefficient type, term order: G = graded, L = lexicographic, big c = using the big coefficients, big e = using the big exponents, s = server JVM.

options, system	JDK 1.5	JDK 1.6
BigInteger, G	16.2	13.5
BigInteger, L	12.9	10.8
BigRational, L, s	9.9	9.0
BigInteger, L, s	9.2	**8.4**
BigInteger, L, big e, s	9.2	**8.4**
BigInteger, L, big c	66.0	59.8
BigInteger, L, big c, s	45.0	**45.8**

There is a simple benchmark for comparing the multiplication of sparse polynomials in [37]. It times the computation of the product of two polynomials $q = p \times (p + 1)$, with integer coefficients, where $p = (1 + x + y + z)^{20}$, with bigger coefficients $p = (10000000001(1 + x + y + z))^{20}$, or with bigger exponents $p = (1 + x^{2147483647} + y^{2147483647} + z^{2147483647})^{20}$. The results for JAS are shown in table 1 and for other systems in table 2. The timings are from [2] and show that JAS is more than 3 times faster than the old MAS system but also 3.5 times slower than the Singular system. However Singular is not able to compute the example with bigger exponents. For this example JAS is 45% faster than Maple, and 65% faster than Mathematica. This shows that JAS (and the Java VM) matches the performance of general purpose systems. For further details see the discussion in [2].

3.9 Applications

As an application of the generic polynomials we have implemented some more advanced algorithms, such as polynomial reduction or Buchbergers algorithm to compute Groebner bases. The algorithms are also implemented for polynomial rings over principial ideal domains and Euclidean domains and for solvable polynomial rings (with left and two-sided variants) and modules over these rings. The performance of these implementations will be covered in a future publication.

Table 2. Polynomial multiplication, other systems

Computing times in seconds on AMD 1.6 GHz CPU and Intel 2.7 GHz.
Options are: coefficient type is rational number for MAS, integer for Singular and unknown for Maple an Mathematica, big c = using the big coefficients, big e = using the big exponents, term order G = graded, L = lexicographic.

options, system	time	@2.7GHz
MAS 1.00a, L, GC = 3.9	**33.2**	
Singular 2-0-6, G	2.5	
Singular, L	**2.2**	
Singular, G, big c	12.95	
Singular, L, big exp	out of memory	
Maple 9.5	**15.2**	9.1
Maple 9.5, big e	19.8	11.8
Maple 9.5, big c	64.0	38.0
Mathematica 5.2	**22.8**	13.6
Mathematica 5.2, big e	30.9	18.4
Mathematica 5.2, big c	30.6	18.2

3.10 Parallelization

JAS has been implemented with the goal of being *thread safe* from the beginning. This is mainly achieved by implementing all algebraic elements by immutable objects. This avoids any synchronization constructs for the methods at the cost of some more object creations. We have, however, not studied the impact of this on the performance.

In the beginning, we had developed some utility classes for easier parallelization of the algorithms. In the mean time some classes are no more required, since equivalent substitutions exist in `java.util.concurrent`. We have replaced some of them in the latest refactorings of the library.

In the `ufd` package there is a nice parallel proxy class, which provides effective employment of the fastest algorithms at *run-time*. In the time of multi-core CPU computers, we compute the gcd with two (or more) different implementations in parallel. Then we return the result from the fastest algorithm, and cancel the other still running one. The gcd proxy can be generated from `GCDFactory.<C>getProxy()`. For example in a Groebner base computation with rational function coefficients, requiring many gcd computations, the fastest was 3610 times the subresultant algorithm and 2189 times a modular algorithm.

3.11 Libraries

The advantage of (scientific) libraries is apparent. Javas [4] success is greatly influenced by the availability of its comprehensive libraries of well tested and efficient algorithms. Also languages like Perl or PHP profit greatly from their comprehensive sets of available libraries. JAS is an attempt to provide a library for polynomial arithmetic and applications. There are other activities in this direction, however they are not all open source projects using the GPL license.

The goal of the jscl-meditor [21] project "is to provide a java symbolic computing library and a mathematical editor acting as a front-end to the former."

jscl has a similar mathematical scope as JAS and we are looking for possibilities to cooperate in future work. The project JScience [23] aims to provide "the most comprehensive Java library for the scientific community". The library has a broader perspective than JAS, in that it wants to support not only mathematics, but also physics, sociology, biology, astronomy, economics and others. There is the Orbital library [22], which provides algorithms from (mathematical) logic, polynomial arithmetic with Groebner bases and optimizations with genetic (*sic*) algorithms. The Apache software foundation distributes a numerical mathematical library as part of the `org.apache.commons` package [38]. It is a "library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the Java programming language".

3.12 Java environment

In [31] we have advocated the usage of standard libraries in favor of special implementations. In earlier computer algebra systems the creators had to implement many standard data structures by themselves. But now, we have the situation, that many of these data structures are available in form of well designed and tuned libraries, like the Java collection framework or the standard template library (STL) from C++. With this approach one can save effort to implement well known data structures again and again. Moreover, one profits from any improvements in the used libraries and improvements of the Java virtual machine (JVM). This has been exemplified by the performance improvements between JDK 1.5 and JDK 1.6 in section 3.8, table 1. Since Java is 64-bit ready we have been able to run Groebner base computations in 50 GB main memory.

4 Conclusions

JAS provides a sound object oriented design and implementation of a library for algebraic computations in Java. For the first time we have produced a type safe library using generic type parameters in a contemporary programming language. With Javas interfaces we are able to implement all algebraic types required for a computer algebra system. The generic programming of Java allows a type safe implementation of polynomial classes. It also helped to drastically reduce code size and facilitates code reuse. Type safety in JAS is only limited by the dependent type problem, which cannot be avoided in popular contemporary programming languages. With checked and unchecked exceptions we can model all necessary algebraic semantics of methods. The recursive `RingElem` and polynomial design allows the implementation of all important multivariate polynomial greatest common divisor algorithms. The usage of design patterns, e.g. the factory pattern for object creation, allows a clean concept for object oriented algebraic programming. Although object orientation looks strange to mathematicians, it is state of the art in modern programming languages. The performance of the library is comparable to general purpose computer algebra systems, but can not match the performance of special hand tuned systems. We have demonstrated that a large portion of algebraic algorithms can be implemented in a convenient Java library, e.g. non-commutative solvable polynomials or greatest common divisors. The parallel and distributed implementations of Groebner base algorithms draw heavily on the Java packages for concurrent

programming and inter-networking. For the main working structures we built on the Java packages for multi-precision integers and the collection framework. The steady improvements of Java and its package implementations, leverage the performance and capabilities of JAS. A problem with Java's type erasure was identified as a general feature of generic object oriented programming languages and is not specific to Java.

In the future we will implement more of 'multiplicative ideal theory', i.e. multivariate polynomial factorization.

Acknowledgments

I thank Thomas Becker for discussions on the implementation of a polynomial template library and Raphael Jolly for the discussions on the generic type system suitable for a computer algebra system. With Samuel Kredel I had many discussions on the C++ type system and implementation choices for algebraic algorithms in C++. Thanks also to Markus Aleksy and Hans-Guenther Kruse.

References

1. Kredel, H.: On the Design of a Java Computer Algebra System. In Gitzel, R., Aleksy, M., Schader, M., Krintz, C., eds.: Proc. PPPJ 2006. ACM International Conference Proceedings Series, Mannheim University Press (2006) 143–152
2. Kredel, H.: On a Java Computer Algebra System, its performance and applications. In: Special Issue of PPPJ 2006, Science of Computer Programming, Elsevier (2008) (in print)
3. Kredel, H.: The Java algebra system. Technical report, http://krum.rz.uni-mannheim.de/jas/ (since 2000)
4. Sun Microsystems, Inc.: The Java development kit. Technical report, http://java.sun.com/, accessed 2007, May (1994-2007)
5. Grabmaier, J., Kaltofen, E., Weispfenning, V., eds.: Computer Algebra Handbook. Springer (2003)
6. Jenks, R., Sutor, R., eds.: axiom The Scientific Computation System. Springer (1992)
7. Bronstein, M.: Sigma$^{\text{it}}$ - a strongly-typed embeddable computer algebra library. In Calmet, J., Limongelli, C., eds.: Proc. DISCO 1996. Volume 1128 of Lecture Notes in Computer Science., Springer (1996) 22–33
8. Watt, S. In: Aldor. in Computer Algebra Handbook, Springer (2003) 265–270
9. Greuel, G.M., Pfister, G., Schönemann, H. In: Singular - A Computer Algebra System for Polynomial Computations. in Computer Algebra Handbook, Springer (2003) 445–450
10. Buchmann, J., Pfahler, T. In: LiDIA. in Computer Algebra Handbook, Springer (2003) 403–408
11. Noro, M., Takeshima, T.: Risa/Asir–a computer algebra system. In: Proc. ISSAC'92, ACM Press (1992) 387–396
12. Kredel, H., Pesch, M. In: MAS: The Modula-2 Algebra System. in Computer Algebra Handbook, Springer (2003) 421–428
13. Gruntz, D., Weck, W.: A Generic Computer Algebra Library in Oberon. Manuscript available via Citeseer (1994)
14. Collins, G.E., Loos, R.G.: ALDES and SAC-2 now available. ACM SIGSAM Bull. **12**(2) (1982) 19
15. Whelan, C., Duffy, A., Burnett, A., Dowling, T.: A Java API for polynomial arithmetic. In: Proc. PPPJ'03, New York, Computer Science Press (2003) 139–144

16. Niculescu, V.: A design proposal for an object oriented algebraic library. Technical report, Studia Universitatis "Babes-Bolyai" (2003)
17. Niculescu, V.: OOLACA: an object oriented library for abstract and computational algebra. In Vlissides, J.M., Schmidt, D.C., eds.: OOPSLA Companion, ACM (2004) 160–161
18. Bernardin, L., Char, B., Kaltofen, E.: Symbolic computation in Java: an appraisement. In Dooley, S., ed.: Proc. ISSAC 1999, ACM Press (1999) 237–244
19. Conrad, M.: The Java class package com.perisic.ring. Technical report, http://ring.perisic.com/, accessed 2006, September (2002-2004)
20. Becker, M.Y.: Symbolic Integration in Java. PhD thesis, Trinity College, University of Cambridge (2001)
21. Jolly, R.: jscl-meditor - java symbolic computing library and mathematical editor. Technical report, http://jscl-meditor.sourceforge.net/, accessed 2007, September (since 2003)
22. Platzer, A.: The Orbital library. Technical report, University of Karlsruhe, http://www.functologic.com/ (2005)
23. Dautelle, J.M.: JScience: Java tools and libraries for the advancement of science. Technical report, http://www.jscience.org/, accessed 2007, May (2005-2007)
24. Musser, D., Schupp, S., Loos, R.: Requirement oriented programming - concepts, implications and algorithms. In: Generic Programming '98: Proceedings of a Dagstuhl Seminar, Springer-Verlag, Lecture Notes in Computer Science 1766 (2000) 12–24
25. Schupp, S., Loos, R.: SuchThat - generic programming works. In: Generic Programming '98: Proceedings of a Dagstuhl Seminar, Springer-Verlag, Lecture Notes in Computer Science 1766 (2000) 133–145
26. Becker, T., Weispfenning, V.: Gröbner Bases - A Computational Approach to Commutative Algebra. Springer, Graduate Texts in Mathematics (1993)
27. Geddes, K.O., Czapor, S.R., Labahn, G.: Algorithms for Computer Algebra. Kluwer (1993)
28. Stansifer, R., Baumgartner, G.: A Proposal to Study Type Systems for Computer Algebra. Technical Report 90-07, Johannes Kepler University, Linz, Austria (1990)
29. Baumgartner, G., Russo, V.F.: Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. Software - Practice and Experience **25**(8) (1995) 863–889
30. Abdali, S.K., Cherry, G.W., Soiffer, N.: An object-oriented approach to algebra system design. In Char, B.W., ed.: Proc. SYMSAC 1986, ACM Press (1986) 24–30
31. Kredel, H.: A systems perspective on A3L. In: Proc. A3L: Algorithmic Algebra and Logic 2005, University of Passau (April 2005) 141–146
32. Meyer, B.: Genericity versus inheritance. In: OOPSLA. (1986) 391–405
33. Poll, E., Thomson, S.: The type system of Aldor. Technical report, Computing Science Institute Nijmegen (1999)
34. Fateman, R.J.: Advances and trends in the design and construction of algebraic manipulation systems. In: Proc. ISSAC 1990, ACM Press (1990) 60–67
35. Dragan, L., Watt, S.: Performance Analysis of Generics in Scientific Computing. In: Proceedings of Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE Computer Society (2005) 90–100
36. Sun Microsystems, Inc.: Improvements to program execution speed. http://java.sun.com/j2se/1.5.0/docs/guide/performance/speed.html, accessed 2007, May 10 (2004)
37. Fateman, R.J.: Draft: Comparing the speed of programs for sparse polynomial multiplication. http://www.cs.berkeley.edu/ fateman/papers/fastmult.pdf, accessed 2007, May 5 (2002)
38. Apache Software Foundation: Commons-Math: The Jakarta mathematics library. Technical report, http://jakarta.apache.org/commons/, accessed 2007, May 18 (2003-2007)