Fostering Interoperability in Java-Based Computer Algebra Software

Heinz Kredel
IT-Center University of Mannheim, Germany
Email: {kredel}@rz.uni-mannheim.de

***Abstract*—This paper considers interoperability in Java based computer algebra software. It is well known that interoperability in Java software is greatly enhanced by simple but expressive interfaces. However, there is no commonly agreed set of interfaces for Java based computer algebra software. When no common interfaces exist it is required to develop adapter classes for each pair of different interfaces to achieve interoperation. We present three existing interfaces from the Java Algebra System (JAS), from JLinAlg and from Apache Commons Math. We discuss advantages and problems with each set of interfaces and define a useful common subset as a proposal for a future standard.**

***Keywords*-interfaces as types, computer algebra software, interoperability of libraries**

I. Introduction

One of the success factors for a software eco-system is the a core set of libraries included with the distribution and the availability of (third-party) libraries for many application areas. These factors are for example fulfilled for GNU-Linux and also for the Java system.

In this paper we study the API design for Java libraries in the area of computer algebra software. We focus on the design of generic interfaces and study three examples: the interfaces found in Apache commons Math, the interfaces found in JLinAlg and the interfaces in the Java algebra system (JAS). The requirements for reusable (Java) libraries can be summarized as follows:

1) **separately compiled library:** No need for recompilation to use it, like a Java jar-file, or a C shared object library, not like C++ templates.
2) **generic and object oriented:** Usable for a (wide) variety of "basic" objects or data types.
3) **statically type safe:** Type errors should be spotted during compilation of the application and at least during run-time.
4) **usable in parallel:** Use immutable objects, ensure thread safety and objects distributable to clusters of computers.

Because of the JVM run-time system with automatic memory management it is possible to attempt to design and build such a library, see e.g. [1].

Generic libraries provide even more advantages, as one library can compute with the data structures of another library. For example a library providing algorithms for linear algebra could be used in a library for general commutative algebra. Different libraries have different focus on mathematical content and useful algorithms. Also the research and development groups do not have the man-power to implement every algorithm in all versions in their library. Such limitations to interoperability are addressed at various levels:

1) **system level:** The OpenMath specifications provide XML interfaces at the highest level [2]. For example (commercial) computer algebra systems like Maple, Mathematica or Matlab are monolithic systems which can interchange information only at the level of the interaction language.
2) **scripting level:** The Sage computer algebra software [3] is written in Python and provides access to several (open source) computer algebra systems written in C or C++. For example to Singular [4], Pari, Gap or Kant. Sage makes use of C/C++ libraries where possible and else uses the interaction langage of the respective system. The different interfaces and APIs of such systems are adapted at the Python level, e.g. objects for the Pari system are lifted to Sage Python objects, then mapped to Singular objects to be used with Singular algorithms.
3) **library level:** This is the topic of this paper. We want to provide a common set of interfaces to interoperatively use the JAS library, [5], [6], [7] the Apache Commons Math library [8] and the JLinAlg library [9].

A. Related Work

The related work published on type systems for computer algebra or abstract data type (ADT) approaches to computer algebra has been summarized in [1], [10]. Type-safe design considerations in computer algebra are mostly centered around the *axiom* computer algebra system and are described, e.g. starting with [11], [12]. See also the work on the *Magma* computer algebra system, e.g. [13]. Type-safe design considerations in computer algebra are described in [14], [15], [11], [16], [17]. Generic programming issues are discussed for example in [18], [19], [20] and the references therein. Further related work is mentioned in the paper as required.

B. Outline of the paper

In section 2 we introduce the interfaces and examples of their implementing classes of the three systems. Then section 3 we discuss advantages of the approaches and propose a common library usable by all systems. The last section 4 draws some conclusions.

II. Interfaces and Example Classes

Each library defines a set of interfaces taylored to the respective needs and design. We concentrate the discussion on the topic of ring respective field interfaces which are central for interoperation.

All interfaces in the discussed projects distinguish between elements of an algebraic structure and factories to create such elements.

A. Apache Commons Math (ACMath)

As the library concentrates on linear algebra it consideres elements from fields and not more general rings. The interfaces are `FieldElement` and `Field` for the factory class, see figure 1. The methods defined are very minimal, only the methods `add()`, `subtract`, `multiply()` and `divide()` for the ring operations `+`, `-`, `*`, `/` are defined. The last method throws an `ArithmeticException` for the case of a divison of zero. The factory only defines the methods `getZero()` and `getOne()`. The intefaces take a type parameter `T`, which is not further restricted.

This minimalistic design is described in [8] as "(Apache Commons Math) emphasizes small, easily integrated components rather than large libraries with complex dependencies and configurations."

If one looks at an implementing class, for example the rational numbers in classes `BigFraction` and the factory `BigFractionField`, there are more interfaces and classes implemented. First, the interfaces `Comparable` and `Serializable` are implemented. Second the class `Number` is extended, which mandates conversion methods, like `intValue()`. In the implementation the methods defined in the interface are overloaded with four different parameter types: the class itself, the class `BigInteger` and the primitive types `int` and `long`. These overloaded methods are, however, not reflected in the interface. Also the methods `pow()` for exponentiation are not reflected in the interface. The selectors `getDenominator()` and `getNumerator()` are not meaningful in the interface. The converting methods `bigDecimal()` should eventually be defined in the interface.

B. JLinAlg

The interface `IRingElement`, see figure 2, for ring elements defines the methods `add()`, `subtract()`, `multiply()` and `divide()`, moreover there is a method `inverse()` to compute inverses in the respective rings.

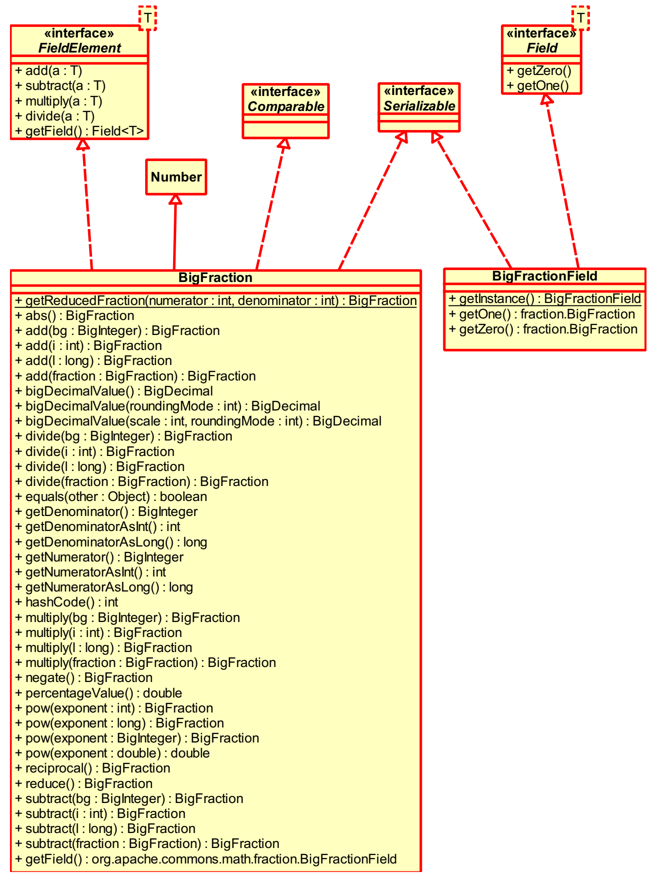

Figure 1. Apache Commons Math Interfaces

Further methods are `negate()` and `abs()` to compute the additive inverse and the absolute value. Aditionally there are predicates `isZero()` and `isOne()`. Then there are additional comparison methods `lt()`, `gt()`, `le()` and `ge()`. Further interessting methods are `norm()` to compute the norm and `apply()` to apply a unary-function of the element.

The interface `IRingElementFactory` for the ring factory defines the methods to create elements, like `one()`, `zero()` and `m_one()` for minus one. Then there are several methods to construct random elements according to several options in the methods `randomValue()` and `gaussianRandomValue()`. Then there are some conversion methods, called `get()`, to convert elements of other types to this ring. Finally there are methods to construct arrays `getArray()` and methods to convert vectors and matricies `convert()`. The interfaces take a type paramter `RE` which is restricted to `IRingElement`.

The interfaces of JLinAlg are accompanied by abstract classes `RingElement` and `RingElementFactory`. In `RingElement` the predicates are implemented with the help of the `equals()` method and the comparisons are implemented by the `compareTo()` method. Further, sub-

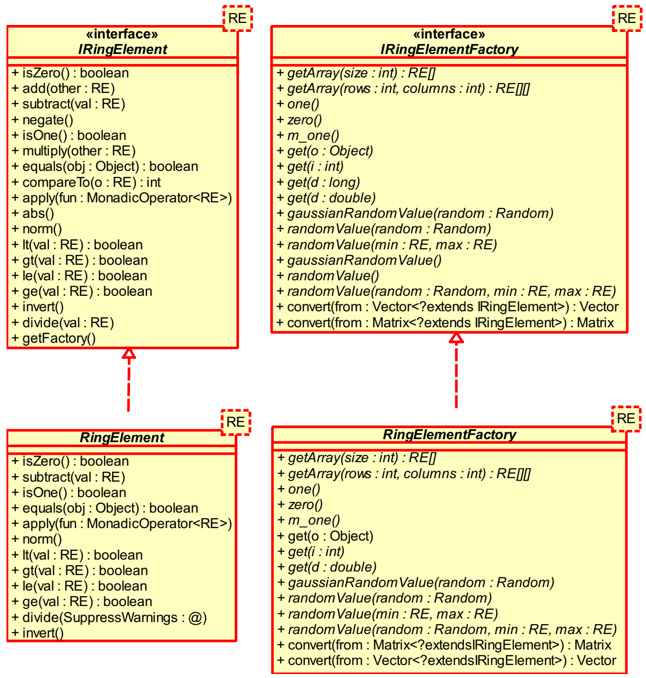

Figure 2. JLinAlg Interfaces

traction is implement with the help of negation and addition. The methods `divide()` and `inverse()` provide implementations which throw exeptions if not overwritten and throw the execption `DivisionByZeroException`. The factory provides a default implementation for `get()` conversion method with the help of a string representation and implement the `convert()` methods with the help of `get()`.

C. Java Algebra Aystem (JAS)

The interfaces of JAS, see figure 3, for rings `RingElem` are extended from more basic interfaces, namely `AbelianGroupElem` and `MonoidElem`, which in turn both extend `Element`. The interfaces are generic and take a type parameter `C` which is restricted to the respective interface.

The `Element` interface is the top of the JAS interface hierarchy and extends itself the Java interfaces `Clonable`, `Comparable` and `Serializable`. Besides the methods mandated by the super interfaces, the `Element` interface defines the method `factory()` to obtain the factory and and `toScript()` to get a string representation which is suitable as input to a scripting language like (J/P)ython and (J)Ruby.

The interface `AbelianGroupElem` defines the commutative additive methods `sum()`, `subtract()`, `negate()` and `abs()`. The predicate `isZERO()` tests if an element is zero and the method `signum()` computes the sign of an element. The interface `MonoidElem` defines the, eventually non-commutative, multiplicative methods `multiply()`, `divide()`, `inverse()` and `remainder()`. The predicate `isONE()` tests of the one element of the ring and `isUnit()` determines if the element is invertible in the ring. The interface `RingElem` extends the just described interfaces and adds two new methods `gcd()` and `egcd()` for the computation of the (extended) greatest common divisor. The interface `FieldElem` extends `RingElem` and is empty, as all required methods are already defined.

The factory interface `ElementFactory` at the top of the JAS factory interface hierarchy defines conversion methods `fromInteger()` from integers and `parse()` from strings. Then there are methods to create random elements, method `random()`, and the generators of the ring, method `generators()`. Method `toScript()` provides a string representation which is suitable as input to a scripting language like (J/P)ython and (J)Ruby. Finally there is a predicate `isFinite()` to test if the respective ring is finite or infinite.

The interface `AbelianGroupFactory` just defines the method `getZERO()` to obtain the neutral element. The interface `MonoidFactory` defines the method `getONE()` to obtain the neutral element. The predicates `isCommutative()` and `isAssociative()` determine if the respective ring is commutative and associative. The interface `RingFactory` defines the predicate `isField()` to determine if the respective ring is already a field. The method `characteristic()` obtains the characteristic of the ring. The `FieldFactory` interface is again empty.

III. Comparison and Proposal

All libraries provide (at least) some generic algorithms, which are written using Java 5 type parameters for "basic" algebraic objects defined to implement an interface. The basic concepts of the designs are not so different. E.g. the set of interfaces of each system uses a split design into the elements of the algebraic structure and a factory object, which are used to generate or construct elements of the structure.

Regarding the size of each set of interfaces the most simple is defined in Apache Commons Math. More elaborate methods are defined in JLinAlg and most comprehensive set is defined in JAS.

The libraries have different targets: Apache commons math focuses on linear algebra over commutative fields of charactristic zero. JLinAlg has its focus also on linear algebra but considers commutative fields of arbitrary characteristic. JAS has its focus on more general commutative and non-commutative (non-linear) algebras.

Remark: The name `add()` for the addition of ring / field elements is not lucky. The ring / field elements are implemented and designed as immutable objects. However, the method `add()` is used in the Java collection framework as a mutable method. It modifies a collection by adding a further object to it. When working with lists of ring / field

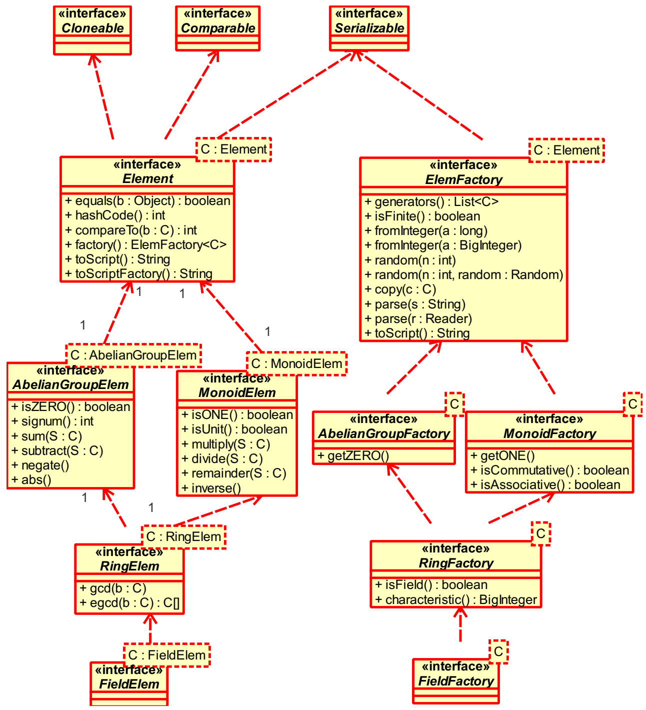

Figure 3. JAS Interfaces

elements it is somewhat likely to confuse the `add()` of a ring element with an `add()` of a list by forgetting to assign the return value of a ring `add()` to a ring variable. As this made up some serious and hard to find bugs in JAS it was decided to use the name `sum()` for this purpose.

There is a trade-off in the number of methods in an interface: More methods in the interface place a burden on the implementor of the classes as more methods must be implemented. Missing methods in the interface will lead to case distinctions at runtime or can even break the generic design of the library.

In this respect the interfaces of ACMath are too limited as they define too few methods. As the implementations of ACMath classes show, they see the need for more methods, like `pow()`, and the need to extend further interfaces and classes from Java. Note, that `pow()` only depends on the field methods, so it could already be implemented in the interface. This is at the moment possible with so called 'traits' in Scala [21] and used in ScAS [22]. This will eventually be also possible in Java 8 with default implementations of interface methods.

JLinAlg defines a more detailed set of methods. For ring elements the difference only in the `norm()` and `apply()` methods. `norm()` is important for complex number structures, together with an `conjugate()` method, and should eventually go to an additional `ComplexElem` interface (or `StarRingElem` as it is called in JAS). JLinAlg also defines more methods for the factories, like the conversions `get()` and `convert()` plus some more elaborate methods for random element generation.

The JAS interfaces provide the most mature selection of definitions which have proven to be required within the last 6 years. JAS started with a small set of method definitions like ACMath, but was extended in the last years as the library growed and more algorithms for various applications became integrated. For example `isFinite()` is required if infinite fields of finite characteristic have to be worked with. `characteristic()` is required if algorithms for finite fields appear in the library. The predicates `isCommutative()`, `isAssociative()` and `isField()` are required in generic algorithms, if the library will not only handle infinite commutative fields. The conversion methods `fromInteger()` and `parse()` are similar to methods from JLinAlg and should eventually be changed to a more general concept. For example with an overloaded method like `valueOf()` for strings, integers or other related elements, e.g. occuring in embeddings.

As it should be possible to transport algebraic ojects over a network to different computers, we recomend that the interfaces should extend the Java interface `Serializable`. So we can ensure interoperation in a distributed environment. We also recomend that the interfaces extend the Java interfaces `Clonable` and `Comparable`. This ensures that the algebraic objects can be used efficiently together with the Java collections framework, for example they can be used as keys in sorted maps.

For the interoperation between JAS and ACMath as well as for the interoperation of JAS and JLinAlg we have written adapter classes. These adaper classes just implement the respective interface and delegate the method call to a native object from the respective adapter pair. For a greater number of systems this is certainly an approach which does not scale well. There is also a run-time overhead for the delegation, however we have not meassured the computing time to study this case in more detail.

So we propose to use revised JAS interfaces as a common base set of interfaces. It provides a proven set of useful methods which allow generic implementations for a wide range of rings. The burden to implement the predicates for rings, is not high as in many cases it should be possible just to return a truth value without further computation. Whether the interfaces should be defined in flat form as in JLinAlg or ACMath or in structured from as in JAS remains to be decided. This set of interfaces could very well have its home at the Apache Commons project.

IV. Conclusions

We have presented the interfaces from three computer algebra libraries, namely from the Java Algebra System (JAS), from JLinAlg and from Apache Commons Math. We discussed the advantages and problems with each set of interfaces and defined a useful subset as a proposal for a future standard.

Acknowledgments

We thank our colleagues Thomas Becker, Wolfgang K. Seiler, Thomas Sturm, Raphael Jolly, Axel Kramer, Sherm Ostrowsky, and others for various discussions on the design of and the requirements for JAS and ScAS.

References

[1] R. Jolly and H. Kredel, "Generic, type-safe and object oriented computer algebra software," in *Proc. CASC 2010*. Springer, LNCS 6244, 2010, pp. 162–177.

[2] OpenMath Consortium, "OpenMath, version 2.0," OpenMath Consortium, Tech. Rep., 2004, http://www.openmath.org/ standard/ om20-2004-06-30/ omstd20html-0.xml, accessed Jan 2010. [Online]. Available: http://www.openmath.org/, accessed Jan 2010

[3] W. Stein, *SAGE Mathematics Software*, The SAGE Group, since 2005, http://www.sagemath.org, accessed Oct 2011.

[4] G. Greuel, G. Pfister, and H. Schönemann, *Singular - A Computer Algebra System for Polynomial Computations*. in Computer Algebra Handbook, Springer, 2003, pp. 445–450.

[5] H. Kredel, "The Java algebra system (JAS)," http://krum.-rz.uni-mannheim.de/jas/, Tech. Rep., since 2000.

[6] ——, "On a Java Computer Algebra System, its performance and applications," *Science of Computer Programming*, vol. 70, no. 2-3, pp. 185–207, 2008.

[7] ——, "Unique factorization domains in the Java computer algebra system," in *Automated Deduction in Geometry*, ser. Lecture Notes in Computer Science, T. Sturm and C. Zengler, Eds. Springer, 2011, vol. 6301, pp. 86–115. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-21046-4_5

[8] Apache Software Foundation, "Commons-Math: The Jakarta mathematics library," http://commons.apache.org/, accessed Nov 2009, Tech. Rep., 2003-2010.

[9] A. Keilhauer, S. D. Levy, A. Lochbihler, S. Ökmen, G. L. Thimm, and C. Würzebesser, "JLinAlg: a Java-library for linear algebra without rounding errors," http:// jlinalg.source-forge.net/, accessed Jan 2010, Tech. Rep., 2003-2010.

[10] R. Jolly and H. Kredel, "Algebraic structures as typed objects," in *Proc. CASC 2011*. Springer, LNCS 6885, 2011, pp. 294–308.

[11] R. Jenks and R. Sutor, Eds., *axiom The Scientific Computation System*. Springer, 1992.

[12] S. Watt, "Aldor," in *Computer Algebra Handbook, Springer*, 2003, pp. 265–270.

[13] W. Bosma, J. J. Cannon, and C. Playoust, "The Magma algebra system I: The user language," *J. Symb. Comput.*, vol. 24, no. 3/4, pp. 235–265, 1997.

[14] H. J. Davenport and B. M. Trager, "Scratchpad's view of algebra I: Basic commutative algebra," in *Proc. DISCO'90*. Springer LNCS 429, 1990, pp. 40–54.

[15] H. J. Davenport, P. Gianni, and B. M. Trager, "Scratchpad's view of algebra II: A categorical view of factorization," in *Proc. ISSAC'91, Bonn*, 1991, pp. 32–38.

[16] H. J. Davenport, "Abstract data types in Computer Algebra," in *Proc. MFCS 2000*. Springer LNCS 1893, 2000, pp. 21–35.

[17] M. Bronstein, "Sigma$^{\mathrm{it}}$ - a strongly-typed embeddable computer algebra library," in *Proc. DISCO 1996*. University of Karlsruhe, 1996, pp. 22–33.

[18] D. Musser, S. Schupp, and R. Loos, "Requirement oriented programming - concepts, implications and algorithms," in *Generic Programming '98: Proceedings of a Dagstuhl Seminar*. LNCS 1766, Springer, 2000, pp. 12–24.

[19] S. Schupp and R. Loos, "SuchThat - generic programming works," in *Generic Programming '98: Proceedings of a Dagstuhl Seminar*. LNCS 1766, Springer, 2000, pp. 133–145.

[20] L. Dragan and S. Watt, "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, pp. 90–100.

[21] Martin Odersky, "The Scala programming language," http:// www.scala-lang.org/, accessed June 2011, Tech. Rep., 2003-2011.

[22] R. Jolly, "Object Scala found - a JSR223-compliant version of the Scala interpreter," in *Scala Days 2011*, 2011, p. to appear.