Generic, Type-safe and Object Oriented Computer Algebra Software

Heinz Kredel and Raphael Jolly

IT-Center, University of Mannheim, Germany and Databeans, Paris, France
kredel@rz.uni-mannheim.de, raphael.jolly@free.fr

Abstract. Advances in computer science, in particular object oriented programming, and software engineering have had little practical impact on computer algebra systems in the last 30 years. The software design of existing systems is still dominated by ad-hoc memory management, weakly typed algorithm libraries and proprietary domain specific interactive expression interpreters. We discuss a modular approach to computer algebra software: usage of state-of-the-art memory management and run-time systems (e.g. JVM) usage of strongly typed, generic, object oriented programming languages (e.g. Java) and usage of general purpose, dynamic interactive expression interpreters (e.g. Python) To illustrate the workability of this approach, we have implemented and studied computer algebra systems in Java and Scala. In this paper we report on the current state of this work by presenting new examples.

1 Introduction

The great success of computer algebra systems like Maple or Mathematica in engineering and science has lead to reduced efforts in the construction of software components suitable for algorithmic and software engineering research. Maple and Mathematica and numerical systems like Matlab, are currently required in the daily work of many engineers and scientists. Therefore, the focus, marketing and presentation of these systems is oriented to the needs of engineers and scientists: tool boxes with strong emphasis on documentation and education. On the other side there is demand for software systems suitable for algorithmic and software engineering research. For this research it is essential to control every aspect of the software. The critique on systems like Mathematica or Maple from this research groups vary from inappropriate or insufficient implementations of algorithms without the ability to repair or extend the code by better implementations [1] to insufficient data type support [2, 3], and requirements for object oriented implementations [4]. The design chosen 30 years ago makes it moreover difficult to evolve these systems according to current needs or ideas [5, 6]. Furthermore, the concepts of Maple and Mathematica are challenged by Google web-application developments [7] in the same way as Microsoft Word is challenged by Google web documents and toolkits [8]. The Eclipse Rich Client Platform [9] is yet another alternative [10]. Additionally the software architecture of these systems may not be suited for multi-CPU and Grid-Computing with 10.000 to 100.000 processing nodes.

Contemporary open source computer algebra software like Singular ([11, 12]), Kant, PariGP, Gap and others partially suffer from similar software architectural origins as Maple or Mathematica. These systems are monolithic, highly integrated software systems with nearly no public interfaces besides the command line shell. The low-level run-time system which provides mainly memory management with automatic garbage collection is tightly coupled with the next level of arithmetic algorithm implementations like integer arithmetic or polynomial arithmetic. This structure is a great obstacle in the reuse of parts of the implementations in other systems or projects. Nevertheless there is a very clever attempt to provide a common interface to some of these systems with the Sage project [13].

In this situation we undertook the experiment to rewrite major portions of a computer algebra system in object oriented programming languages. We could have joined efforts with Axiom/Aldor to a certain extent for our object oriented approach. However, our two main motivations to use Java and Scala are: first, to explore to which extent these programming languages are realy suited for this task and second, have platforms which support Cloud computing (e.g. [14]) and Smart devices (e.g. [15]).

1.1 Related work

There is not much work published on object oriented programming for algebraic or symbolic algorithm implementation. There is more work published on type systems for computer algebra or abstract data type (ADT) approaches to computer algebra. The main question is the expressiveness required to implement algebraic algorithms. However, the requirements for libraries and interactive parts are constantly mangled and not cleanly separated as in our approach. A first paper on CAS with SmallTalk [16] and the ongoing work of the Axiom developers can to some extend be viewed as object oriented [3]. Newer approaches start with [17] in Common Lisp, then using C++ [1, 18]. Early considerations of Java in computer algebra and symbolic computation [19–21]. Newer approaches using Java are [22, 23] or [24, 25], or our approaches starting with [26, 27]. An object oriented but non-Java approach to computer algebra exists as part of the Focalize project [28]. Type-safe design considerations in computer algebra are described in [29, 30, 3, 31, 32, 17]. Generic programming issues are discussed for example in [33–35] and the references therein. Interoperability via OpenMath is discussed in [36–40]. Further thoughts on the future of computer algebra systems see [6, 5]. Further related work is mentioned in the paper as required, and in the section on future work.

1.2 Outline

In section 2 we discuss design considerations for object oriented computer algebra and symbolic computation software. Examples for the construction of such systems are presented in section 3. Section 4 shows some future work and the final section draws some conclusions.

2 Design considerations

The proposed software architecture builds on other software projects as much as possible. Only the parts specific to computer algebra are to be implemented. We identify three major parts for computer algebra software.

- run-time infrastructure with memory management,
- statically typed object oriented algorithm libraries,
- dynamic interactive scripting interpreters.

We discuss the first points in the following subsections. For the third point see our articles [41, 42].

2.1 Run-time systems

Run-time systems with *automatic memory management* can be taken from virtual machines, like the Java JVM or the .NET CLR. Advantages of virtual machines:

- constant maintenance and improvements,
- more opportunities for code optimization with just-in-time compilers,
- memory management with automatic garbage collection,
- exception and security constraint handling,
- independence of computer hardware and optimization requirements,
- suitable for multi-CPU and distributed computing.

Disadvantages of virtual machines are the dependency on the chosen platform.

Software development in computer algebra must deal with unpredictable dynamic memory requirements of the algebraic and symbolic objects occurring in a computation, so a crucial necessity is the availability of automatic memory management in the run-time system.

2.2 Object oriented software

The second building block for the design and implementation of computer algebra systems, is *object oriented programming methodology*. It can be characterized as follows

- usage of contemporary (object oriented) software engineering principles,
- modular software architecture, consisting of
  - usage of existing implementations of basic data structures like integers or lists
  - generic type safe algebraic and symbolic algorithm libraries
  - thread safe and multi-threaded library implementations
  - algebraic objects transportable over the network
- high performance implementations of algorithms with state of the art asymptotic complexity but also fast and efficient for small problem sizes,

– minimizing the 'abstraction penalty' which occurs for high-level programming languages compared to low-level assembly-like programming languages.

Our research in this new direction is the design and the construction of object oriented computer algebra libraries in Java and Scala, called JAS and ScAS. The main concepts and achievements have recently been presented and published in a series of computer science and computer mathematics conferences [27, 43–50] and on the projects Web-pages [51, 26].

3 Examples

In this section we discuss some examples of the object oriented approach to computer algebra software. First we discuss the design of the basic interfaces and classes for algebraic structures, namely rings and polynomials. We present the ScAS design, as the similar JAS design has been presented elsewhere. In the next sub-section, we discuss the design of algorithms for polynomial factorization. For the performance of Java implementations of multivariate polynomial arithmetic, greatest common divisor computation and comprehensive Gröbner bases construction see [45, 47, 48]. Problems of object oriented programming for which we have not found satisfactory solutions yet will be covered in a subsequent publication. Some knowledge of Java and Scala is required for the understanding of this section.

3.1 Ring elements and polynomials

The type-safe design of the basic structural interfaces in ScAS makes use of "traits", the Scala equivalent of Java's interfaces (i.e. with multiple inheritance), but with the added feature that some methods can be implemented, thus taking code re-use a step further. These traits are type-parametrized, which provides modularity while forbidding arithmetic operations on incompatible types at compile time. The root of the hierarchy is the trait `Element`. Each trait in the hierarchy comes with a `Factory` which is used to obtain new instances of its type and do various operations on these.

```
object Element {
  trait Factory[T <: Element[T]] {
    def random(numbits: Int)(
        implicit rnd: scala.util.Random): T
  }
}
trait Element[T <: Element[T]] extends Ordered[T] { this: T =>
  val factory: Element.Factory[T]
  def equals(that: T) = this.compare(that) == 0
  def ><(that: T) = this equals that
  def <>(that: T) = !(this equals that)
}
```

The notation `this: T =>` is called a self-type and is used to specify the future type of "this" which is currently not known as the type is abstract and cannot be instantiated. The definitions of `><` and `<>` must be made because the operators `==` and `!=`, which in Scala are normally routed to `equals`, are not type-parametrized and have an argument of type `Any`. Next we have traits for commutative (abelian) additive groups, (multiplicative) semi-groups, and monomials. Scala provides operator overloading, resulting in a natural mathematical notation.

```
object AbelianGroup {
  trait Factory[T <: AbelianGroup[T]] extends Element.Factory[T] {
    def zero: T
  }
}
trait AbelianGroup[T <: AbelianGroup[T]]
    extends Element[T] { this: T =>
  override val factory: AbelianGroup.Factory[T]
  def isZero = this >< factory.zero
  def +(that: T): T
  def -(that: T): T
  def unary_+ = this
  def unary_- = factory.zero - this
  def abs = if (signum < 0) -this else this
  def signum: Int
}
```

Some methods like `abs` can already be implemented in terms of other ones which remain abstract for now (`signum`, `-`). If `signum` is meaningful in the respective ring depends on the ring. It is a design decision, explained for example in [47], to let interfaces define methods which may fail in certain rings.

```
trait SemiGroup[T <: SemiGroup[T]] extends Element[T] { this: T =>
  def *(that: T): T
}
```

`Semigroup` has no corresponding `Factory` as there is no additional feature compared to `Element`. Thus `Monoid.Factory` below will inherit directly from `Element.Factory`, whereas `Monoid` inherits from `SemiGroup`:

```
object Monoid {
  trait Factory[T <: Monoid[T]] extends Element.Factory[T] {
    def one: T
  }
}
trait Monoid[T <: Monoid[T]] extends SemiGroup[T] { this: T =>
  override val factory: Monoid.Factory[T]
  def isUnit: Boolean
  def isOne = this >< factory.one
  def pow(exp: BigInt) = {
```

```
    assert (exp >= 0)
    (factory.one /: (1 to exp.intValue)) {
      (l, r) => l * this
    }
  }
}
```

The method `isOne` tests exactly for the 1 in the ring, `isUnit` tests if the element is an associate of 1 (whence if it is invertible). `pow` is implemented here, which will save these lines of code everywhere Monoid will be inherited. Note, such elegant solutions are not possible with Java interfaces and Java abstract classes are also of no help in such situations. Binary exponentiation is not shown to simplify the example.

```
object Ring {
  trait Factory[T <: Ring[T]]
      extends AbelianGroup.Factory[T] with Monoid.Factory[T] {
    def characteristic: BigInt
  }
}
trait Ring[T <: Ring[T]] extends AbelianGroup[T]
    with Monoid[T] { this: T =>
  override val factory: Ring.Factory[T]
}
```

`Ring` inherits multiply from `AbelianGroup` and `Monoid`, and declares a `characteristic`. Below we outline the implementation of a `Polynomial` class from the basic structures we have just defined.

```
object Polynomial {
  class Factory[C <: Ring[C]](val ring: C,
      val variables: Array[Variable],
      val ordering: Comparator[Int])
      extends Ring.Factory[Polynomial[C]] {
    def generators: Array[Polynomial[C]]
    def apply(value: SortedMap[Array[Int], C])
        = new Polynomial(this)(value)
    override def toString: String
  }
}
class Polynomial[C <: Ring[C]](
    val factory: Polynomial.Factory[C])(
    val value: SortedMap[Array[Int], C])
    extends Ring[Polynomial[C]] {
  def elements: Iterator[Pair[Array[Int], C]]
  def headTerm = elements.next
  def degree: Int
```

```
  def isUnit = this.abs.isOne
  override def toString: String
}
```

To be really accurate, the above Polynomial definition in ScAS is abstract and is in fact a trait, just like Ring from which it inheritates. This is be able to sublclass it to a SolvablePolynomial (i.e. non-commutative) in addition to a regular Polynomial. The abstract Polynomial has a self-type parameter like the other abstract types, in order to preserve type-safety. The definitions are then sketched as follows:

```
object Polynomial {
  trait Factory[T <: Polynomial[T, C],
                C <: Ring[C]] extends Ring.Factory[T] {
    def multiply(w: T, x: Array[Int], y: C) = { // commutative
    }
  }
}
trait Polynomial[T <: Polynomial[T, C], C <: Ring[C]]
      extends Ring[T]

object SolvablePolynomial {
  trait Factory[T <: Polynomial[T, C], C <: Ring[C]]
        extends Polynomial.Factory[T, C] {
    override def multiply(w: T, x: Array[Int], y: C) = {
                 // non-commutative case
    }
  }
}
```

The various mechanisms above allow multiple options to be combined mostly independently with minimal code duplication. Among the possible dimensions of parametrization, we have implemented:

- the coefficient type (`C`, above)
- the underlying data structure (array, list, tree) of the polynomial, involving the type of the "value" field (`SortedMap[Array[Int], C]` above) which can be parametrized trough a Scala abstract type member
- the type of the exponents (`P` in ScAS code, not shown)
- we have made a prototype implementation of polynomials with different algorithms for gcd computation : the type of the polynomial (factory) tells whether the gcd algorithm is of simple, primitive, or subresultant kind. This is intended as an improvement on JAS, see the next section for how this is implemented there.
- whether or not the polynomial is a `SolvablePolynomial`

We are studying the implementation for:

– an improved parametrization of the type of the exponents. It is typically `Int` or `Long`, so this is a Java primitive type. Unlike Java, Scala allows to parametrize over these. But this is currently made through boxing and unboxing, with performance impact. Scala 2.8 will provide "type specialization" whereby a class with a primitive type as parameter will be implemented in an optimized way.
– the list of variables and the ordering. Currently it is possible to arithmetically mix polynomials in different sets of variables, which is not optimal. Parameterizing this item requires some kind of dependent type, as described in section 7.3 of [47].

3.2 Unique factorization domains

To further exemplify the usefulness of object oriented computer algebra software development we studied the implementation of some non-trivial examples for library design. We limit the discussion to algorithms for multivariate polynomials in (constructive) unique factorization domains. Examples, like Gröbner bases, comprehensive Gröbner bases, univariate power series and elementary integration of rational functions are discussed elsewhere [51, 49, 48]. For the mathematical background see some text books on the topic, like [52]. We give an overview of the respective interfaces and classes of JAS and then compare it to the approach of Scratchpad (now Axiom) [29, 30].

Figure 1 shows an UML overview diagram of the involved interfaces and classes for the computation of greatest common divisors of multivariate polynomials. In case an algorithm is only meaningful for univariate polynomials, the respective implementations convert the polynomials to univariate polynomials in the main variable with multivariate coefficients. There are implementations for such recursive representations, which are not shown. We start with an interface `GreatestCommonDivisor`. It defines the method names for a ring with gcd algorithm. First there is the method `gcd()` itself, together with the method `lcm()` to compute the least common multiple of two polynomials. With the help of `gcd()` the algorithms for the content `content()` and the primitive part `primitivePart()` computation can be implemented. The methods `coPrime()` compute lists of co-prime polynomials from given lists of polynomials.

The abstract super class for the implementations is called `GreatestCommonDivisorAbstract`. It implements nearly all methods defined in the `GreatestCommonDivisor` interface. The abstract methods are `baseGcd()` and `recursiveUnivariateGcd()`. The method `gcd()` first checks for the recursion base, and eventually calls `baseGcd()`. Otherwise it converts the input polynomials to recursive representation, as univariate polynomials with multivariate polynomial coefficients, and calls method `recursiveUnivariateGcd()`.

The concrete implementations come in two flavors. The first flavor implements only the methods `baseGcd()` and `recursiveUnivariateGcd()`, using the setup provided by the abstract super class. There are implementations for various polynomial remainder sequence (PRS) algorithms: simple, monic, primitive

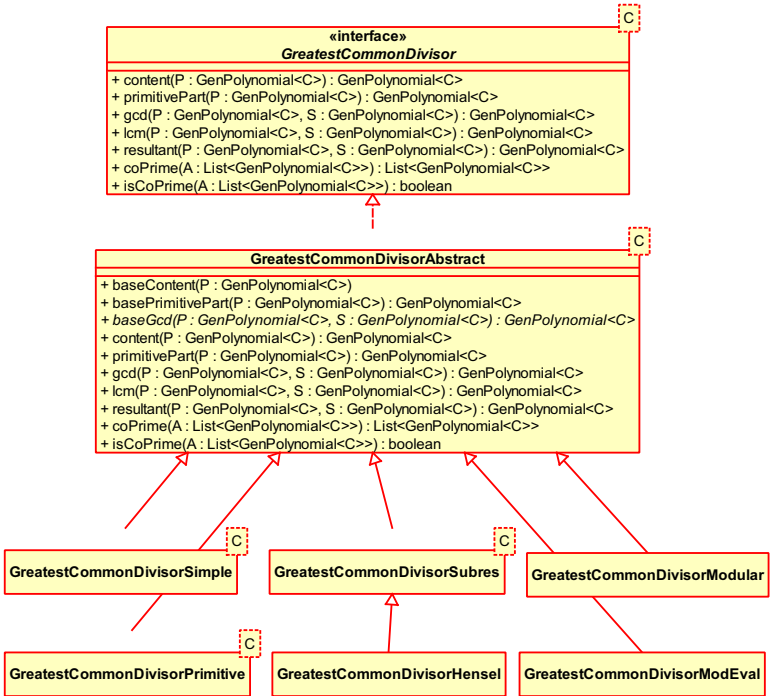

Fig. 1. Greatest common divisor classes

and the sub-resultant algorithm (in the respective classes). These implementations are generic for any (UFD) coefficient ring. The second flavor directly implements `gcd()` without providing `baseGcd()` and `recursiveUnivariateGcd()`. The algorithms compute gcds first modulo some suitable prime numbers and then interpolate the result using Chinese remainder algorithms, or in the Hensel case via powers of the prime number. The later versions are only valid for integer or modular coefficient classes `BigInteger`, `ModInteger` or `ModLong`.

An overview for the classes for the squarefree decomposition of multivariate polynomials is shown in figure 2. Again, there is an interface `Squarefree` and an abstract class `SquarefreeAbstract`. The other classes are for coefficients from rings or fields of characteristic zero, and for finite and infinite fields of characteristic $p > 0$. Method `squarefreeFactors()` of the interface decomposes a polynomial into squarefree parts. Method `squarefreePart()` computes the squarefree part of a polynomial. Method `isSquarefree()` test the respective property for a polynomial. `isFactorization()` tests if a given map or list is actually a squarefree decomposition for a given polynomial. The interface has additionally methods `coPrimeSquarefree()` to compute lists of co-prime and squarefree polynomials from given lists of polynomials.

The abstract class `SquarefreeAbstract` implements most of the methods specified in the interface. There are four methods which remain abstract and

Fig. 2. Squarefree decomposition classes

have to be implemented in sub-classes. For multivariate polynomials the methods `squarefreeFactors()` and `squarefreePart()` and for univariate polynomials the methods `baseSquarefreeFactors()` and `baseSquarefreePart()`. The implementation has to distinguish three major and two sub cases: coefficient fields of characteristic zero, class `SquarefreeFieldChar0`, and coefficients not from a field `SquarefreeRingChar0`. For coefficient fields of characteristic non-zero, class `SquarefreeFieldCharP`, we need two sub-classes for finite and infinite fields, classes `SquarefreeFiniteFieldCharP` and `SquarefreeInfiniteFieldCharP`.

Figure 3 shows the class layout of an interface, two abstract and several concrete classes for multivariate polynomial factorization. The interface `Factorization` defines the most useful factorization methods. The method `factors()` computes a complete factorization with no further preconditions and returns a `SortedMap`, which maps polynomials to the exponents of the polynomials occurring in the factorization. The method `factorsSquarefree()` factors a squarefree polynomial. It returns a list of polynomials since the exponents will all be 1. Method `factorsRadical()` computes a complete factorization, but returns a list of polynomials, that is, all exponents are removed. Methods `isIrreducible()` and `isReducible()` test the respective properties for a polynomial. `isFactorization()` tests if a given map is actually a factorization for a given polynomial.

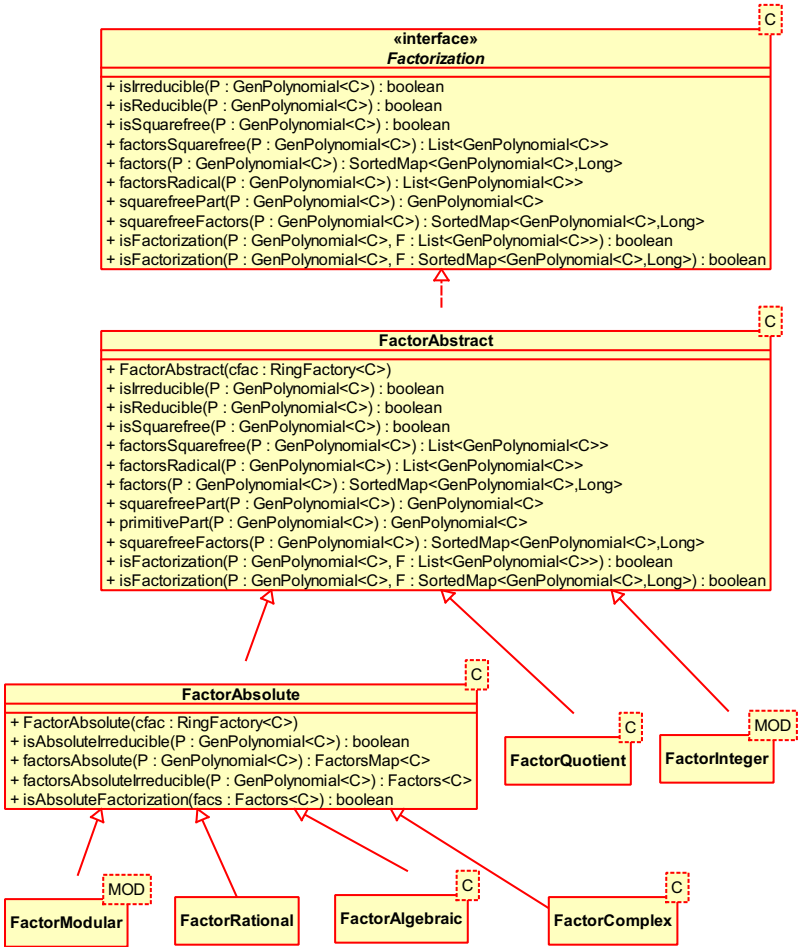

Fig. 3. Polynomial factorization classes

The abstract class `FactorAbstract` implements all of the methods specified in the interface. Only `baseFactorSquarefree()` for the factorization of a square-free univariate polynomial is declared abstract and must be implemented for each coefficient ring. For multivariate polynomials Kronecker's algorithm is used to reduce this case to a univariate problem and to reassemble multivariate factors from univariate ones. This algorithm is not particularly fast and will be accompanied by a multivariate Hensel algorithm in the future.

The absolute factorization of a polynomial, that is, a factorization over an algebraically closed coefficient field, is implemented in class `FactorAbsolute`. The main method is `baseFactorsAbsoluteIrreducible` for the factorization of a polynomial which is irreducible over the given coefficient field. This method constructs a field extension of the ground field from the given irreducible poly-

nomial. The given polynomial is then factored in this algebraic field extension with the algorithms from class `FactorAlgebraic`. Then there are classes which implement factorization for particular coefficient rings, for example modular integers, integers and rational numbers as well as generic algorithms for algebraic numbers and quotients of polynomials.

We come to a comparison with the "categorical view of factorization" in Scratchpad and Axiom. There factorization is attached to the generic univariate polynomial 'class', named `SparseUnivariatePolynomial`. The 'method' called `factorPolynomial()` computes the factorization of a given polynomial. 'Categorical' then means, that it will compute a factorization of multivariate polynomials over any coefficient ring which has itself a constructive implementation of factorization of univariable polynomials over itself.

The building blocks for factorization in JAS, the computation of greatest common divisors and the squarefree decomposition are fully generic in this sense (except the special implementations for integers and integers modulo primes). They only require coefficient rings which themselves have constructive greatest common divisors (for univariate polynomials over itself). Also for squarefree decomposition all cases for coefficient fields of characteristic zero, respectively finite and infinite fields of characteristic $p > 0$ are implemented generically.

Multivariate polynomial factorization reduction to univariate polynomial factorization is fully generic by the Kronecker substitution algorithm. Univariate polynomial factorization for specific coefficient rings, namely `BigInteger`, `ModInteger` and `BigRational` is obviously not generic. However, `FactorAlgebraic` and `FactorQuotient` is generic for coefficients rings which allow constructive univariate polynomial factorization. The selection of appropriate factorization algorithms (as well as squarefree decomposition and gcd algorithms) is provided in class `FactorFactory` (respectively `SquarefreeFactory` and `GCDFactory`) by method `getImplementation(RingFactory r)`. It takes a factory for the coefficient ring `r` as parameter and returns a suitable implementation (if one is available or exists [30]). The returned implementation is of type `FactorAbstract<C>` which must at least contain a method to factor univariate polynomials over the ring `r`. For example we can factor polynomials with coefficients from the ring $\mathbb{Q}(\sqrt{2})(x)(\sqrt{x})$. In summary we have shown, that very general approaches like the categorical approach of Scratchpad can be implemented in a generic, typesafe way in Java.

4 Future work

With the described software stack, computer algebra software can make use of many modern computing and presentation environments. For example, using existing rich client platforms like Eclipse by MathEclipse [10]. Or for using webservice and Cloud computing platforms, like Google App Engine by Symja [53]. This approach is moreover interesting as it has a parser for the Mathematica programming language. A further example is computer algebra on portable devices: we have a version of the JSCL library compiled for the Palm platform [26],

and we plan to use the scripting ability of Android to make computer algebra available on this system [54].

The JVM infrastructure enables moreover a new lower level of interaction between computer algebra systems. It complements the most general high level interoperation between computer algebra systems using OpenMath [37] and the intermediate level interoperation using Python like the Sage project [13]. OpenMath builds on very general interfaces defining algebraic and symbolic expressions whereas Sage takes an agnostic approach and completely ignores common interfaces for the integrated computer algebra systems.

Examples for a tight interoperation on the JVM are the linear algebra library of Apache Commons Math [55] or the linear algebra library JLinAlg [56]. Using adaptors for ring elements it is possible to use generic implementations of linear algebra algorithms with JAS algebraic objects and vice-versa. The various adaptor classes between generic computer algebra libraries could be avoided if we could define (and agree on) a common interface for ring elements and implement it in each system.

A more loose integration is based on the Java scripting framework specified in JSR 223 [57] and was outlined in jscl-meditor to combine several scripting engines accessed from a common mathematical editor [26]. Further interoperation of computer algebra systems is investigated with a Maxima port to ABCL common lisp on the JVM [58]. There is also a Reduce on the JVM [59, 21] which could be accessed via JSR 223 (with some effort in code rearrangement). Combining OpenMath and Java is studied in Popcorn and Wupsi [36, 39, 40, 37, 38].

5 Conclusions

We have shown how computer algebra software can be designed and implemented leveraging 30 years of advances in computer science. Our approach concentrates on mathematical aspects by re-using software components developed by other projects. Namely, the Java programming language together with the JVM runtime system and interactive scripting languages, for example Python. The JVM infrastructure opens new ways of interoperability of computer algebra systems on Java byte-code level. This infrastructure gives also new opportunities to provide computer algebra software on new computing devices, software as a service, distributed or cloud computing. Our object oriented approach with the Java and Scala programming languages makes it possible to implement non-trivial algebraic structures in a type-safe way which can be stacked and plugged together in unprecedented ways.

Acknowledgments

We thank Thomas Becker for discussions on the implementation of a polynomial template library. JAS itself has improved by requirements from Axel Kramer and Georg Thimm. We thank our colleagues W. K. Seiler, Dongming Wang and Thomas Sturm for various discussions and for encouraging our work. Thanks also to the referees for the suggestions to improve the paper.

References

1. Frink, A., Bauer, C., Kreckel, R.: Introduction to the GiNaC framework for symbolic computation within the C++ programming language. J. Symb. Comput. (2002)
2. Stansifer, R., Baumgartner, G.: A Proposal to Study Type Systems for Computer Algebra. Technical Report 90-07, Johannes Kepler University, Linz, Austria (1990)
3. Jenks, R., Sutor, R., eds.: axiom The Scientific Computation System. Springer (1992)
4. Calmet, J., Seiler, W.M.: Computer algebra and field theories. Mathematics and Computers in Simulation **45** (1998) 33–37
5. Watt, S.M.: On the future of Computer Algebra Systems at the threshold of 2010. In: Proceedings ASCM-MACIS 2009, Kyushu University, Fukuoka, Japan (2009) 422–430
6. Wolfram, S.: WolframAlpha. Technical report, http://www.wolframalpha.com/, accessed Jan 2010 (2009)
7. Certik, O.: SymPy Python library for symbolic mathematics. Technical report, http://code.google.com/p/sympy/ accessed Nov 2009 (since 2006)
8. GWT Developers: Google Web Toolkit consists of a Java - to - JavaScript compiler, user interface API, and related tools. Technical report, http://code.google.com/webtoolkit/ accessed Nov 2009 (2008)
9. Eclipse Developers: Eclipse rich client platform (RCP). Technical report, http://www.eclipse.org/ accessed Nov 2009 (2008)
10. Kramer, A.C.: MathEclipse is usable as an online Java computer algebra system or Eclipse plugin. Technical report, http://www.matheclipse.org/ accessed Nov 2009 (2009, since 2002)
11. Greuel, G., Pfister, G., Schönemann, H. In: Singular - A Computer Algebra System for Polynomial Computations. in Computer Algebra Handbook, Springer (2003) 445–450
12. Greuel, G., Pfister, G.: A SINGULAR introduction to commutative algebra. Springer Verlag (2007)
13. Stein, W.: SAGE Mathematics Software (Version 2.7). The SAGE Group. (since 2005) http://www.sagemath.org, accessed Nov 2009.
14. AppEngine Developers: Google App Engine enables you to build and host web apps on the same systems that power Google applications. Technical report, http://code.google.com/appengine, accessed June 2010 (2010)
15. Android Developers: Android is a software stack for mobile devices including an operating system, middleware and key applications. Technical report, http://code.google.com/android/ accessed Nov 2009 (2008)
16. 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
17. Zippel, R.: Weyl computer algebra substrate. In: Proc. DISCO '93, Springer-Verlag Lecture Notes in Computer Science 722 (2001) 303–318
18. Parisse, B.: Giac/Xcas, a free computer algebra system. Technical report, University of Grenoble (2008)
19. Bernardin, L., Char, B., Kaltofen, E.: Symbolic computation in Java: an appraisement. In Dooley, S., ed.: Proc. ISSAC 1999, ACM Press (1999) 237–244
20. Bernardin, L.: A Java framework for massively distributed symbolic computing. SIGSAM Bull. **33**(3) (1999) 20–21

21. Norman, A.C.: Further evaluation of Java for symbolic computation. In: ISSAC '00: Proc. International Symposium on Symbolic and Algebraic Computation 2000, ACM (2000) 258–265
22. Niculescu, V.: A design proposal for an object oriented algebraic library. Technical report, Studia Universitatis "Babes-Bolyai" (2003)
23. Niculescu, V.: OOLACA: an object oriented library for abstract and computational algebra. In: OOPSLA Companion, ACM (2004) 160–161
24. 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
25. Platzer, A.: The Orbital library. Technical report, University of Karlsruhe, http://www.functologic.com/ (2005)
26. Jolly, R.: jscl-meditor - Java symbolic computing library and mathematical editor. Technical report, http://jscl-meditor.sourceforge.net/, accessed Nov 2009 (since 2003)
27. Kredel, H.: A systems perspective on A3L. In: Proc. A3L: Algorithmic Algebra and Logic 2005, University of Passau (April 2005) 141–146
28. Focalize Developers: Focalize is a software distribution for program certification. Technical report, http://focalize.inria.fr/ accessed Jun 2010 (2005-2010)
29. Davenport, H.J., Trager, B.M.: Scratchpad's view of algebra I: Basic commutative algebra. In: Proc. DISCO'90, Springer LNCS 429 (1990) 40–54
30. Davenport, H.J., Gianni, P., Trager, B.M.: Scratchpad's view of algebra II: A categorical view of factorization. In: Proc. ISSAC'91, Bonn. (1991) 32–38
31. Davenport, H.J.: Abstract data types in Computer Algebra. In: Proc. MFCS 2000, Springer LNCS 1893 (2000) 21–35
32. Bronstein, M.: Sigma$^{\text{it}}$ - a strongly-typed embeddable computer algebra library. In: Proc. DISCO 1996, University of Karlsruhe (1996) 22–33
33. Musser, D., Schupp, S., Loos, R.: Requirement oriented programming - concepts, implications and algorithms. In: Generic Programming '98: Proceedings of a Dagstuhl Seminar, LNCS 1766, Springer (2000) 12–24
34. Schupp, S., Loos, R.: SuchThat - generic programming works. In: Generic Programming '98: Proceedings of a Dagstuhl Seminar, LNCS 1766, Springer (2000) 133–145
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. Freundt, S., Horn, P., A.Konovalov, S.Linton, D.Roozemond: Symbolic computation software composability protocol (SCSCP) specification, version 1.3. Technical report, SCIEnce Consortium (2009)
37. OpenMath Consortium: OpenMath, version 2.0. Technical report, OpenMath Consortium (2004) http://www.openmath.org/ standard/ om20-2004-06-30/ omstd20html-0.xml, accessed Jan 2010.
38. SCIEnce Consortium: Symbolic computation infrastructure for Europe. Technical report, SCIEnce Consortium (2009)
39. Horn, P., D.Roozemond: The Popcorn OpenMath representation, version 1.0. Technical report, SCIEnce EU Project (2009)
40. Horn, P., D.Roozemond: WUPSI universal Popcorn SCSCP interface, version 1.0. Technical report, SCIEnce EU Project (2009)
41. Jolly, R., Kredel, H.: How to turn a scripting language into a domain specific language for computer algebra. Technical report, http://arXiv.org/ abs/0811.1061 (2008)

42. Jolly, R., Kredel, H.: Symbolic script programming for Java. Technical report, http://arXiv.org/ abs/0906.2315 (2009)
43. Kredel, H.: On the Design of a Java Computer Algebra System. In: Proc. PPPJ 2006, University of Mannheim (2006) 143–152
44. Kredel, H.: Evaluation of a Java Computer Algebra System. In: Proceedings ASCM 2007, National University of Singapore (2007) 59–62
45. Kredel, H.: Evaluation of a Java computer algebra system. Lecture Notes in Artificial Intelligence **5081** (2008) 121–138
46. Kredel, H.: Multivariate greatest common divisors in the Java Computer Algebra System. In: Proc. Automated Deduction in Geometry (ADG), East China Normal University, Shanghai (2008) 41–61
47. Kredel, H.: On a Java Computer Algebra System, its performance and applications. Science of Computer Programming **70**(2-3) (2008) 185–207
48. Kredel, H.: Comprehensive Gröbner bases in a Java Computer Algebra System. In: Proceedings ASCM 2009, Kyushu University, Fukuoka, Japan (2009) 77–90
49. Kredel, H.: Distributed parallel Gröbner bases computation. In: Proc. Workshop on Engineering Complex Distributed Systems at CISIS 2009, University of Fukuoka, Japan (2009) CD–ROM
50. Kredel, H.: Distributed hybrid Gröbner bases computation. In: Proc. Workshop on Engineering Complex Distributed Systems at CISIS 2010, University of Krakow, Poland (2010) CD–ROM
51. Kredel, H.: The Java algebra system (JAS). Technical report, http://krum.rz.uni-mannheim.de/jas/ (since 2000)
52. Geddes, K.O., Czapor, S.R., Labahn, G.: Algorithms for Computer Algebra. Kluwer (1993)
53. Kramer, A.C.: Symja a symbolic math system written in Java based on the Math-Eclipse libraries. Technical report, http://code.google.com/p/symja/ accessed Jan 2010 (since 2009)
54. Android Scripting Developers: Android Scripting brings scripting languages to android. Technical report, http://code.google.com/p/android-scripting/ accessed Jun 2010 (2009)
55. Apache Software Foundation: Commons-Math: The Jakarta mathematics library. Technical report, http://commons.apache.org/, accessed Nov 2009 (2003-2010)
56. Keilhauer, A., Levy, S.D., Lochbihler, A., Ökmen, S., Thimm, G.L., Würzebesser, C.: JLinAlg: a Java-library for linear algebra without rounding errors. Technical report, http://jlinalg.sourceforge.net/, accessed Jan 2010 (2003-2010)
57. Sun Microsystems, Inc.: JSR 223: Scripting for the Java platform. Technical report, http://scripting.dev.java.net/, accessed Nov 2009 (2003-2006)
58. ABCL Developers: Armed bear common lisp (ABCL) - common lisp on the JVM. Technical report, http://common-lisp.net/project/armedbear/, accessed Jan 2010 (2003-2010)
59. Reduce Developers: REDUCE interactive system for general algebraic computations. Technical report, http://www.reduce-algebra.com/, accessed Jan 2010 (1968-2010)